RICE UNIVERSITY

gNek: A GPU Accelerated Incompressible Navier Stokes Solver

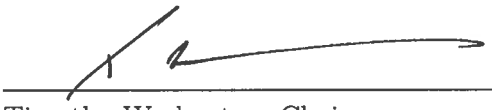
by

Nichole Stilwell

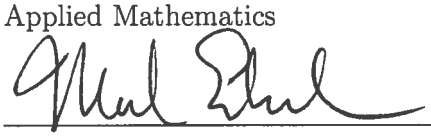
A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Arts

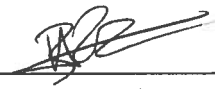
Approved, Thesis Committee:



Timothy Warburton, Chair
Associate Professor of Computational and Applied Mathematics



Mark Embree
Professor of Computational and Applied Mathematics



Beatrice Rivière
Associate Professor of Computational and Applied Mathematics

Houston, Texas

April, 2013

ABSTRACT

gNek: A GPU Accelerated Incompressible Navier Stokes Solver

by

Nichole Stilwell

This thesis presents a GPU accelerated implementation of a high order splitting scheme with a spectral element discretization for the incompressible Navier Stokes (INS) equations. While others have implemented this scheme on clusters of processors using the Nek5000 code, to my knowledge this thesis is the first to explore its performance on the GPU. This work implements several of the Nek5000 algorithms using OpenCL kernels that efficiently utilize the GPU memory architecture, and achieve massively parallel on-chip computations. These rapid computations have the potential to significantly enhance computational fluid dynamics (CFD) simulations that arise in areas such as weather modeling and aircraft design procedures. I present convergence results for several test cases including channel, shear, Kovasznay, and lid-driven cavity flow problems, which achieve the proven convergence results.

Acknowledgments

I would like to thank the members of my committee, Dr. Timothy Warburton, Dr. Mark Embree, and Dr. Beatrice Rivière for their helpful suggestions and encouragements. I would like to especially thank my adviser, Dr. Timothy Warburton for his brilliant computational and mathematical skills and insight. I would also like to thank Dr. Jan Hewitt for her diligent help with the thesis writing.

I thank my friends, both in the CAAM department and from my Lifegroup family, for their support and prayers the last two years. Finally, I thank my family for their neverending love and advocacy. I would not be where I am today without my parents and my sister.

Contents

Illustrations

Tables

Nomenclature

$(\cdot,\cdot)_\Omega$ Inner product over entire domain, page 3

$(\cdot,\cdot)_{D^k}$ Inner product over mesh element, page 3

$\hat{\mathbf{W}}$ Diagonal matrix whose entries are the GLL weights and Jacobian values, page 36

$\hat{\phi}_{ijk}$ Local tensor product basis function, page 28

$\mathbf{A}$ Stiffness matrix that arises in SCP problem, page 36

$\mathbf{D}$ Discrete divergence operator, page 35

$\mathbf{L}$ Discrete Laplace operator, page 35

$\mathbf{W}$ Diagonal matrix whose entries are the GLL weights, page 35

ν Viscosity, page 2

Ω The domain, page 3

ϕ One dimensional basis function, page 28

D^k An element in the mesh, page 3

J Transformation Jacobian, page 30

M_G^{-1} Galerkin inverse mass matrix, page 36

R_e Restriction matrix defining overlap for each preconditioning subdomain, page 44

Re Reynolds number, page 2

Chapter 1

Introduction

This thesis presents a graphics processing unit (GPU) accelerated solver that implements a high order splitting scheme for a spectral element solution of the incompressible Navier Stokes (INS) equations. The solver, gNek, adapts the major algorithms of the program Nek5000, which implements this high-order splitting scheme on CPU like processors, for GPU implementation. This thesis discusses efficient GPU implementaitons of the incompressible Navier Stokes equations using a temporal splitting method and spatial spectral method discretization. This particular scheme was proposed in 1991 by Karniadakis, Israeli and Orszag [30] and has been analyzed and proven to be both an efficient and effective method for numerically solving the incompressible Navier Stokes equations (see for example, Orszag et al. [40], Guermond and Shen [25], Guermond et al. [24]). While this scheme has been implemented on clusters of processors (see, for example [15], [26], [19]), this thesis, to my knowledge, is the first to explore its performance on the GPU. By efficiently utilizing the GPU memory architecture, gNek provides for rapid parallel computation, which has the potential to significantly enhance computational fluid dynamics (CFD) simulations. These CFD simulations arise in many applications, such as weather modeling, ocean modeling, airflow modeling, and aircraft design optimization procedures. This thesis

validates gNek using several benchmark test cases for incompressible Navier Stokes solvers. Specifically, convergence results are presented for channel, shear, Kovasznay, and lid-driven cavity flow problems.

1.1 Problem Description

The incompressible Navier Stokes equations describe the movement of incompressible fluid substances, such as water, and are used to model fluid flow problems:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)\, u = -\nabla p + \frac{1}{Re}\Delta u \tag{1.1a}$$

$$\nabla \cdot u = 0\,. \tag{1.1b}$$

A specific example would be an ocean model that simulates ocean circulation [37]. In these equations, u is the velocity, t is the time, and p is the pressure. The Reynolds number is defined as $Re := \frac{UL}{\nu}$, a dimensionless number that is used to characterize fluid flow as laminar or turbulent, where ν is the kinematic viscosity. In the following, we refer to $(u \cdot \nabla)\, u$ as the advection term, $-\nabla p$ as the pressure gradient term, and $\frac{1}{Re}\Delta u$ as the diffusion term. The $\nabla \cdot u = 0$ term in Equation (1.1b) is the incompressibility constraint on the velocity.

We generalize the strong form of the Navier Stokes equations in Equation (1.1) to the weak form so that we can discretize the problem and then use a variational method, such as the finite element method for the momentum equations. That is, we

discretize the domain, Ω, into elements D^k, such that

$$\Omega = \bigcup_{k=1}^{K} D^k,$$

where K is the total number of elements. In this case the variational problem is to find a velocity, $u \in X_N$, and pressure, $p \in Y_N$, that satisfy

$$\begin{aligned} (\nu \nabla u, \nabla v)_\Omega + \left(\frac{\partial u}{\partial t}, v\right)_\Omega - (p, \nabla \cdot v)_\Omega &= (f, v)_\Omega \\ (\nabla \cdot u, q)_\Omega &= 0 \\ u = g \quad \text{and} \quad \frac{\partial p}{\partial n} &= h \quad \text{on} \quad \partial\Omega, \end{aligned} \tag{1.2}$$

$\forall\, (v, q) \in X_N \times Y_N$, where we define $X_N := \left[\{v \in L^2(\Omega), \text{s.t.}\, v|_{D^k} \in \mathbb{P}_N(D^k)\} \cap H_0^1(\Omega)\right]^3$ and $Y_N := \{q \in L^2(\Omega), \text{s.t.}\, q|_{D^k} \in \mathbb{P}_{N-2}(D^k)\}$ [16]. Here, $(u, v)_{D^k} = \int_{D^k} uv \,\mathrm{d}x \,\mathrm{d}y \,\mathrm{d}z$, for a mesh element D^k. To solve this variational problem, this thesis uses a specific numerical method, namely a temporal splitting method implemented in the context of a spatial spectral element discretization.

1.2 Splitting Method

In order to solve the incompressible Navier Stokes equations, this thesis discretizes the problem in time through a splitting or projection method, and discretizes in space through a spectral element method. In an early contribution, Alexandre Chorin introduced a splitting scheme in 1968 [7]. Chorin presented a way to decouple the velocity and pressure computations, which is the main advantage in using projection methods. Decoupling the system into a velocity solve and pressure solve breaks a

large coupled system into two smaller systems, where one can solve for a vector field (velocity) and a scalar field (pressure). Since 1968, many researchers (for example, [31], [21], [47], [46]) have taken an interest in splitting methods, and several variations of the original method proposed by Chorin have been developed.

The work in this thesis uses the splitting method proposed by Karniadakis, Israeli, and Orszag [30], in which the authors present a high-order splitting method for the incompressible Navier Stokes equations. This method was chosen for implementation on the GPU because it is a highly parallelizable method. This method can be broken into three main steps, namely the Advection, Pressure, and Diffusion steps. These names come from the fact that each step accounts for the contribution from one of the terms in Equation (1.1) to the final velocity at each time step. In particular, at each time step, this method discretizes in time and introduces two intermediate velocities, $\tilde{u}$ and $\tilde{\tilde{u}}$. Using these intermediate velocities, one can solve for the pressure and then ensure that the velocity remains divergence free, or incompressible. In the following equations, u^n is the velocity at time t^n, where $t^n = n * dt$, n is the time step index.

1. Advection Step:

$$\begin{aligned} \frac{\tilde{u} - u^n}{dt} &= -(u^n \cdot \nabla)u^n \\ \Rightarrow \quad \tilde{u} &= u^n - dt(u^n \cdot \nabla)u^n \end{aligned} \tag{1.3}$$

2. Pressure Step

$$\begin{aligned} \frac{\tilde{\tilde{u}} - \tilde{u}}{dt} &= -\nabla p^{n+1} \\ \Rightarrow \quad \tilde{\tilde{u}} &= \tilde{u} - dt\nabla p^{n+1} \end{aligned} \tag{1.4}$$

3. Diffusion Step:

$$\begin{aligned} \frac{u^{n+1} - \tilde{\tilde{u}}}{dt} &= \nu \Delta u^{n+1} \\ \Rightarrow \quad \Delta u^{n+1} - Re u^{n+1} &= -Re \tilde{\tilde{u}} \end{aligned} \tag{1.5}$$

As one can see, these steps decouple Equation (1.1) into solving for the velocity field and the pressure field.

In [30], Karniadakis et al. use an explicit Adams-Bashforth scheme to approximate the advection term, and an implicit Adams-Moulton scheme to approximate the diffusion term. In their semi-discrete formulation, the authors ensure that one can choose the pressure term in a way that enforces a weak incompressibility constraint on the intermediate velocity, $\tilde{\tilde{u}}$. The authors discuss two assumptions that ensure this incompressibity at time $n+1$: the incompressibility condition on the intermediate velocity ($\nabla \cdot \tilde{\tilde{u}} = 0$) and that $\tilde{\tilde{u}}$ satisfies the designated Dirichlet condition in the direction normal to the boundary. These assumptions, which are also made in this thesis, allow one to solve for the pressure as the solution of an elliptic equation with Neumann boundary conditions derived for second order accuracy in time [30].

This choice of boundary conditions for the elliptic pressure equation came from Gresho and Sani [23] and Orszag et al. [40], who presented reasons to enforce Neumann boundary conditions while solving the Poisson equation for pressure. In [40], Orszag et al. were able to show that in the pressure step, (Equation (1.4)) a derived Neumann boundary condition for the pressure leads to the correct pressure calculation. The authors also present a method for computing these Neumann pressure boundary conditions. Specifically, solving the momentum equation normal to the

boundary for the pressure is the properly derived Neumann boundary condition for the pressure step [40].

Further, in [23], Gresho and Sani show that this derived Neumann boundary condition gives a unique solution for $t \geq 0$. On the other hand, using Dirichlet boundary conditions for the pressure works only in the case of $t > 0$. The authors also show that the resultant solution in either case will also be a solution using the other boundary condition, as long as one enforces the Neumann boundary condition at $t = 0$ [23]. In 1995, E and Liu also confirmed this choice for pressure boundary conditions. The authors compare different choices for the numerical boundary condition for the pressure, and in the end, their analysis shows that the Neumann condition is the most favorable [11]. Thus, this thesis uses the derived Neumann boundary conditions in the pressure step of Equation (1.4).

After determining the proper pressure boundary conditions, it is important to examine the convergence of the splitting method as a whole. The convergence results for fractional step methods is the topic of many publications. Guermond, Minev and Shen [24] published a survey of pressure-correction, velocity-correction, and consistent splitting methods in 2006. The paper presents theoretical and numerical results for the convergence of each method. The authors provide error bounds for both velocity and pressure terms of each scheme, and then reference the paper in which these bounds were proven.

In their survey paper, Guermond et al. [24] reference two papers in which Shen

presented error estimates for both first and second order projection methods for the Navier Stokes equations ([49], [50]). In the first paper, Shen analyzes the classical projection methods, both the method proposed by Chorin [7] and the expansions proposed by others, for example, Kim and Moin [31]. In this analysis, Shen provides a derivation of the error estimates for the velocity and pressure of first order schemes used to solve the Navier Stokes equations [49]. In the second paper, Shen similarly provides error estimates for second order schemes using Dirichlet boundary conditions, and also derives a priori estimates for the velocity and pressure [50].

Specific to the work in this thesis, in 2003 Guermond and Shen introduced a velocity-correction projection methods to solve the incompressible Navier Stokes equations [25]. In their paper, Guermond and Shen showed that the splitting method proposed by Karniadakis et al. is equivalent to the rotational form of their velocity-correction method, and provided the first robust proof of the stability and convergence [25]. In particular, the authors show that for the rotational velocity correction method, and consequently the method used in this thesis, the error in the solution for the velocity is $\mathcal{O}(dt)$.

1.3 Spectral Element Methods

The splitting method discussed in the previous section is used as the time discretization to solve the incompressible Navier Stokes equations. In this section, I review the spectral element spatial discretization. Spectral methods have been shown to be

accurate and useful numerical domain decomposition methods in analyzing and modeling fluid flows ([36], [5]). Similar to finite element methods, this method partitions the domain into elements, for which this thesis uses non-overlapping hexahedral elements. Then, one approximates the velocity and pressure fields using tensor product high order polynomial expansions in each element [5]. This thesis uses a high-order spectral element method because of the $\mathcal{O}(N^4)$ asymptotic operator evaluation cost and their excellent wave propogation properties.

This thesis uses an unstructured mesh of hexahedral elements, for which the element connectivity and node information are known. The variational formulation of the incompressible Navier Stokes equations requires the computation of integrals on each element (see Equation (1.2). In order to both precompute these integrals and avoid computing on every single element, a reference element is introduced, which is a cube centered at the origin. To do this, one must also introduce an (r, s, t) coordinate system, and then use a mapping between the reference element (r, s, t) and the mesh elements (x, y, z). This allows one to perform computations on the reference element once and then map the result to each mesh element. This map can be written as

$$\begin{aligned}\begin{pmatrix} x(r,s,t) & y(r,s,t) & z(r,s,t) \end{pmatrix}^T = \frac{1}{8}\Big[&(1-r)(1-s)(1-t)p_1^k + (1+r)(1-s)(1-t)p_2^k \\ &+ (1+r)(1+s)(1-t)p_3^k + (1-r)(1+s)(1-t)p_4^k \\ &+ (1-r)(1-s)(1+t)p_5^k + (1+r)(1-s)(1+t)p_6^k \\ &+ (1+r)(1+s)(1+t)p_7^k + (1-r)(1+s)(1+t)p_8^k\Big],\end{aligned}$$

where $p_i^k = (x_i^k, y_i^k, z_i^k)$, and is the x, y, z coordinate of the i^{th} vertex on the k^{th}

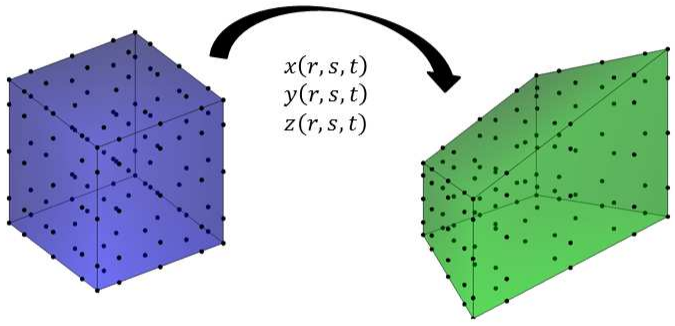


Figure 1.1 : Mapping from the reference element where $\{(r,s,t) \in [-1,1]^3\}$ to the mesh element. Gauss-Lobatto-Legendre nodes are also pictured.

element (see figure 1.1). This mapping uses a change of variables from the reference coordinate system to the mesh coordinate system, which introduces the Jacobian in the computation and mapping of the integrals:

$$\int_{D^k} f \,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z \;=\; \int_{\widehat{D}} f\, J \,\mathrm{d}r\,\mathrm{d}s\,\mathrm{d}t$$

for some function f, where D^k is the mesh element and $\widehat{D}$ is the reference element, and the Jacobian is

$$J = \frac{\partial x}{\partial r}\frac{\partial y}{\partial s}\frac{\partial z}{\partial t} + \frac{\partial x}{\partial s}\frac{\partial y}{\partial t}\frac{\partial z}{\partial r} + \frac{\partial x}{\partial t}\frac{\partial y}{\partial r}\frac{\partial z}{\partial s} - \frac{\partial x}{\partial t}\frac{\partial y}{\partial s}\frac{\partial z}{\partial r} - \frac{\partial x}{\partial s}\frac{\partial y}{\partial r}\frac{\partial z}{\partial t} - \frac{\partial x}{\partial r}\frac{\partial y}{\partial t}\frac{\partial z}{\partial s}.$$

After establishing the mapping, the next step is to choose the basis functions, nodes, and quadrature rule. These choices are based on the first spectral element

method to solve the incompressible Navier Stokes equations, which was proposed by Patera in 1984 [41]. In particular, Patera proposed representing the velocity as a high-order Lagrangian interpolant at Chebyshev points in each element, then treating the advection term with an explicit scheme at these points, while treating the pressure and diffusion terms implicitly [41].

Patera's original formulation of the spectral element method used Gauss-Lobatto-Chebyshev points for the quadrature [41], and in 1988 Rönquist suggested that a better choice would be Gauss-Lobatto-Legendre points, as it provides a lumpable setup of the mass matrix [48]. Several others have also suggested this choice of basis, for example, Korczak and Patera in 1986 [32], along with Maday and Patera [36] and Karniadakis in 1989 [29]. Rönquist specifically suggested using Lagrangian polynomials at Gauss-Lobatto-Legendre points and using Gauss-Lobatto-Legendre quadrature to compute the integrals at each point. Using these nodes not only enables easier assertion of continuity, but also minimizes the coupling between elements, resulting in a diagonal and symmetric mass matrix [48]. Thus, this thesis uses Lagrangian polynomials with Gauss-Lobatto-Legendre points and quadrature (the Gauss-Lobatto-Legendre nodes are pictured in figure 1.1). After establishing the splitting and spectral element scheme used to solve the incompressible Navier Stokes equations, the next step is to discuss a particular problem that arises in the scheme, namely the screened Coulomb potential problem.

1.4 Screened Coulomb Potential

In the splitting method chosen for this thesis, both the pressure and diffusion steps require one to solve the screened Coulomb potential equation, which is

$$(\lambda - \Delta)\, u = -f. \tag{1.6}$$

In this thesis, the pressure step of Equation (1.4) requires one to solve for the pressure at that time step in order to find the second intermediate velocity, $\tilde{\tilde{u}}$. That is, the pressure in Equation (1.4) is the solution to the screened Coulomb potential equation

$$\frac{\nabla \cdot \tilde{u}}{dt} = (\Delta - \lambda)\, p, \tag{1.7}$$

where $\lambda = 0$. To solve Equation (1.6), again we solve the equivalent weak formulation

$$(\nabla u, \nabla v)_\Omega - (\lambda u, v)_\Omega = (f, v)_\Omega\,, \quad \forall v \in V,$$

for a test space V, such that

$$v(x) = 0, \quad x \in \partial\Omega.$$

Solving this system presents a computationally expensive problem, for which this thesis implements the preconditioned conjugate gradient method. This thesis uses an additive Schwarz preconditioner based on finite element methods to solve the Poisson equation, which was studied in this context by Dryja and Widlund in 1992 [10], and Fischer in 1997 [13]. In his paper, Fischer develops a solver for sparse linear systems, specifically the pressure problem that arises in the Uzawa decoupling of the velocity and pressure for the solution to the Navier Stokes equations.

In particular, Fischer demonstrated an additive Schwarz preconditioner that uses overlapping subdomains and a coarse grid projection which can be applied to the inner Gauss points [13]. In his paper, Fischer uses a hexahedral mesh, Gauss-Lobatto-Legendre basis, and a small overlap, which uses only a few GLL nodes outside each element. Similar to the work of Fischer, in 2000 Pavarino presented indefinite overlapping Schwarz methods for both discretizations using mixed finite elements and mixed spectral elements [42]. These methods also solve a small overlapping subdomain problem and a coarse subdomain problem using a hexahedral mesh and GLL nodes [42].

In addition, Pavarino and Warburton introduced an overlapping Schwarz method for unstructured spectral elements in 2000 [43]. In this paper, Pavarino and Warburton present an overlapping Schwarz method for hybrid spectral element discretizations. In this paper, the authors use the entire neighboring element for the overlap in the smaller subdomain elliptic solve and then a larger coarse grid solve. They implement this method in the spectral element code NekTar, and present numerical results for this implementation [43].

Following the well understood Schwarz techniques, this thesis uses a an overlapping Schwarz method as a preconditioner for the conjugate gradient method used to solve the screened Coulomb potential equation in both the pressure (Equation (1.7)) and diffusion (Equation (1.5)) steps.

This preconditioning tool has been comprehensively analyzed. The combination

of spectral elements and finite element precondtitioners was proposed by Orszag in 1980 [39], and since then others have further combined these with additive Schwarz methods, for example, Pavarino and Widlund in 1996 [44]. Cai et al. examined an overlap restriction for this method [4] and Fischer et al. [17] generalized this overlapping Schwarz method to three-dimensional incompressible flows in 1999 [17].

1.5 Parallel Implementation

After outlining the numerical method used to solve the incompressible Navier Stokes equations, it is imperative to discuss the way to implement this method. The structure of this splitting method is inherently parallel, because we perform the same operation on each node in each element. Thus this method lends itself well to parallel implementation. That is, in some instances the computations can simultaneously be performed on multiple processors. The parallelization of spectral element solutions for the Navier Stokes equations has been implemented by Fischer on several different architectures (for examples see [14]). In his early work with Patera, he performed the computations on the Intel vector hypercube [18], and with Rönquist, on the 512 node Intel Delta machine at Caltech [19].

These platforms, the Intel vector hypercube and Intel Delta machine, are clusters of central processing units (CPUs). Each CPU has a certain number of cores, which make up the nodes of these clusters, where the parallel computations take place. In 1992, Fischer and Patera presented a parallel spectral element method along with

timing and speed-up results based on mesh size and number of processors used in the computations [18]. In 1994, Fischer and Rönquist were able to get a fourfold reduction in solution time for a specific boundary layer calculation in three dimensions by parallelizing their algorithm and using mulitple processors for computations [19].

More recently, in 2006, Hamman et al. presented a parallel implementation of a spectral element channel flow solver [26]. In this paper, the authors describe the splitting method proposed by Karniadakis et al. (the method used in this thesis) and then explain the parallelization of each step in the method. This particular implementation is done on several thousand CPU type processors and gives scalability results that point toward implementation at high Reynolds numbers [26].

As technology progresses, there is a growing interest in the parallel implementation of these Navier Stokes solvers, not only on thousands of processors, but on a single workstation. GPUs were designed to be highly parallel devices that allow for simultaneous processing of large blocks of data. Their name, graphics processing units, refers to their original use of creating graphics and video games (see Figure 1.2). Similar to the clusters of CPUs described above, GPUs have multiple cores, which can perform computations in parallel. Thus, using the cores of the GPUs, similar to the nodes of the clusters, to perform the parallel computations will allow for these workstation implementations.

In recent work, Thibault and Senocak presented a CUDA implementation of the projection algorithm presented by Chorin, while using a finite difference method to

Figure 1.2 : Example of a first person video game, which makes use of GPUs to create the images seen by the user.

discretize the advection and diffusion terms [53]. The authors provide both performance and timing results for the GPU implementation, when compared to serial implementation of their solver, and achieves 100 core performance from using four GPUs over a single core processor [53].

In 2009, Göddeke et al. presented an acceleration of an existing finite element solver, known as FEAST (Finite Element Analysis and Toolkit), using GPUs [22]. The authors presented results for stationary laminar flow, where they focus on the linear Navier Stokes equations first, because the solution of the linear subproblems usually takes the most computation time. They also examine the full Navier Stokes solver and compare computation times between using CPUs and using GPUs [22].

The acceleration of solving linear equations was also explored by Krüger and Westermann in 2003, when they presented a framework for applying linear algebra

operators using GPUs [34]. In verifying their framework, Krüger and Westermann use the two dimensional Navier Stoke equations as a test case. They use the GPU to explicitly compute the advection and diffusion terms, and then implement the conjugate gradient method for the pressure solve, and gained a 12-15 times speed up in computations from the CPU to the GPU [34].

Then, in 2011, Brandvik and Pullan developed a three dimensional GPU accelerated Navier Stokes solver [3]. Specifically, the solver built on previous CPU code, the Denton codes, to solve the Navier Stokes equations for flows in turbomachines. In their paper, Brandvik and Pullan use a finite volume method to discretize the Navier Stokes equations, and then introduce a turbulence term for implementation specific to turbomachines [3]. In the implementation, the authors use MPI and CUDA to program processor clusters and GPUs, respectively. The authors also present weak scaling results for the implementation and validation through a turbine test case [3].

Similar to Brandvik and Pullan, this thesis has developed a GPU accelerated version of the Navier Stokes solver Nek5000. Nek5000 is a parallel Navier Stokes solver written in Fortran and C that uses MPI to communicate between CPU-like processors. The development of Nek5000 began in the mid-1980s, when Fischer, Ho, and Rönquist developed the program NEKTON, which was one of the first three dimensional spectral element codes [45]. This code was then commercialized in 1996 as NEKTON 2.0, and then in the mid-1990s, Fischer created a research version of NEKTON 2.0, known as Nek5000, which is a highly scalable version of NEKTON 2.0.

In 1999, Fischer and Tufo were awarded the Gordon Bell Prize in High Performance Computing (HPC) for the quality of its algorithms and parallel performance [45].

Nek5000 uses spectral element and multigrid preconditioning methods to solve the Navier Stokes equations, and runs at scale on over 250,000 CPU processors, or approximately one million cores [45]. Specifically, Nek5000 uses a high-order spectral element spatial discretization and a high-order semi-implicit time discretization. The spectral element multigrid solver is used to find the pressure at each time step, and is scalable. That is, for the pressure solve, Nek5000 employs a coarse-grid solver and a local overlapping subdomain solve. Fischer, Lottes, and the other developers of Nek5000 have taken other steps toward enhancing the scalability of Nek5000, such as a scalable partitioner and a custom all-to-all communication implementation [14]. These efforts culminated in the simulation pictured in Figure 1.3, which is the first spectral element method calculation to use more than one million elements and one billion gridpoints. This simulation of coolant flow is a type of channel flow, where each pin is wrapped in wire, and the length of each channel is $\sim 90000h$, with h being half the channel height.

Other simulations from Nek5000 include hairpin vortex shedding. Figure 1.4 illustrates the simulation of airflow over a hemispherical portrusion, with Reynolds number equal to 700. This flow problem was studied by Acarlar and Smith in 1987 [1], and in this simulation, Nek5000 uses the definition of a vortex as described by Jeong and Hussain [28].

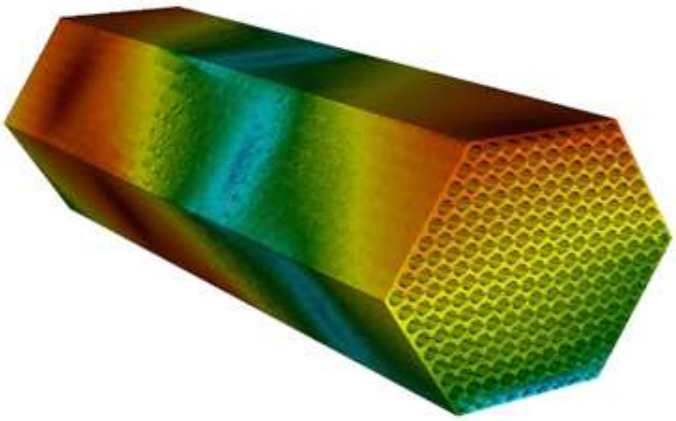


Figure 1.3 : Coolant flow in a 217-pin wire-wrapped subassembly, computed on 32,768 processors using 2.95 million SEM elements of order $N = 7$, and ~ 1 billion grid points [14].

This thesis, in an effort to implement the algorithms in Nek5000 using the fine grain parallelism of the GPU, also uses a high order spectral element spatial discretization and semi-implicit time discretization, which are described in detail in Sections 2.1 and 2.2. Thus far, we have implemented each of the splitting steps using OpenCl kernels. In addition, kernels are used to implement a matrix-free preconditioned conjugate gradient method described in Section 2.3.1 to solve the screened Coulomb potential problem that arises in the Pressure and Diffusion steps of Equations (1.4) and (1.5). Further, gNek uses kernels to implement the pressure Neumann boundary conditions that arise in the SCP problem of Pressure step (1.4).

Figure 1.4 : Simulation for $Re = 700$, computed on the 512-node Intel Paragon at Caltech, using 1021 elements of order $N = 13$, and 2.2 million gridpoints [14].

1.6 Verification

In verifying solvers for the incompressible Navier Stokes equations, there are several canonical test cases. This thesis first uses channel flow, flow between nonslip, parallel boundaries. The exact solution for channel flow can be derived, and then used to check the error between the solver's approximate solution and the expected solution. Second, this thesis examines shear flow, which has a nonzero boundary condition on one wall of a channel. Again, the exact solution for shear flow can be derived. Both of these derivations are given in Chapter 4.

The third test case is known as Kovasznay flow. In 1948, Kovasznay derived an exact solution to the incompressible Navier Stokes equations in two dimensions [33]. This solution is steady state, and thus allows one to test a solver in a simple case. The author describes this flow as a representation of the wake of a two-dimensional

grid [33].

Another test case was proposed by Ethier and Steinman, which is a solution to the three-dimensional incompressible Navier Stokes equations [12]. Additionaly, each of the velocity components depends on all three dimensions and are such that the convective, pressure, and diffusive terms in the Navier Stokes equations are nonzero [12]. Thus, this paper and the solutions presented can be used to test an incompressible Navier Stokes solver in a non-trivial case. That is, one can test a solver with a three-dimensional, time-dependent solution.

Chapter 2 of this thesis discusses the method used to solve the incompressible Navier Stokes equations. Specifically, it explains the use of the mesh, nodes, and quadrature in the spectral element method, and then details each step of the splitting method. Chapter 3 explains the implementation of the chosen method. That is, it thoroughly explains each of the kernels used in the code and relates them back to each part of the method. Finally, Chapter 4 provides numerical results for the test cases previously discussed and also for other more complicated examples, to include three-dimensional, unsteady flow implementations.

Chapter 2

Method

As stated in Chapter 1, gNek implements a splitting method for the spectral element solution to the incompressible Navier Stokes equations (1.1). That is, we use a spectral element method as the numerical technique to solve the variational problem in Equation (1.2), where at each time step, we decouple the pressure and velocity terms. This decoupling is done by splitting the Navier Stokes equation in (1.1a) into an advection, pressure and diffusion step [30]. In this chapter, we first describe the details of the spectral element method. Specifically, we outline the mesh and connectivity information, choice of nodes and numbering scheme, the basis functions and associated geometric factors, and the quadrature rule used to compute integrals. Then, inserting the spectral element spacial discretization into Equation (1.2), we describe the fully discrete splitting scheme. That is, we present the details behind the computation of each step of the splitting method in Equations (1.3) - (1.5) to solve the weak formulation of the incompressible Navier Stokes equations (1.2).

2.1 Spectral Element Method

2.1.1 Mesh and Connectivity Information

The first part of this spectral element method is partitioning the domain, Ω, into a mesh using unstructured, hexahedral elements. That is, we create elements $D^k, k = 1, 2, ..., K$, where K is the total number of elements, such that the subdomains cover Ω:

$$\Omega = \bigcup_{k=1}^{K} D^k.$$

The coordinate information for the elements is used to create three matrices, `gVX`, `gVY`, `gVZ` $\in \mathbb{R}^{vertices \times 1}$, that list the x, y, and z coordinates of each vertex, respectively. Note that *vertices* is the total number of unique vertices, or global vertices, in the meshed domain Ω_h. Next, the element to vertex information is given in `gEToV` $\in \mathbb{N}^{K \times Nverts}$, where $Nverts = 8$, and is the number of vertices on each element. In `gEToV`, the (i, j) entry is the global vertex number of the j^{th} vertex on the i^{th} element. From the mesh information, we also identify which faces of each element, if any, are on the boundary, $\partial\Omega$, of the domain. This information is stored in a matrix, `bcType` $\in \mathbb{N}^{Nfaces \times K}$, where $Nfaces = 6$, and is the number of faces on each element.

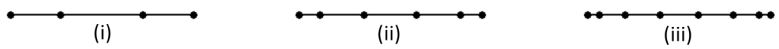
After identifying the necessary vertex and face information, the connectivity information for the mesh is found. These hexahedral elements connect along faces. In other words, if two elements are connected, they share one face, four edges, and four vertices. From the element to vertex information, we create an element to element

connectivity matrix, $\texttt{gEToE} \in \mathbb{R}^{K \times Nfaces}$. In this case, the (i, j) entry is the global element number that connects to the j^{th} face of the i^{th} element. If the entry $(i, j) = 0$, then that particular face is on the boundary, and does not connect to another element. Further, $\texttt{gEToF} \in \mathbb{R}^{K \times Nfaces}$ details the element-to-face information. That is, the (i, j) entry in `gEToF` is the face number that connects to the j^{th} face of the i^{th} element. Once the mesh and connectivity information has been gathered, the next step is to identify the node information.

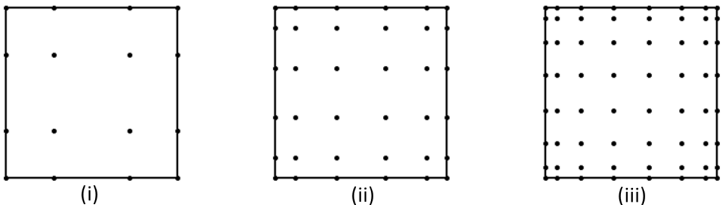
2.1.2 Node Information

This thesis, following [48], uses Gauss-Legendre-Lobatto (GLL) nodes and quadrature to numerically compute the integrals on each element (Section 2.1.6 explains the details of the GLL quadrature rule). Though other node choices exist, such as Gaussian nodes, the GLL scheme places nodes at the endpoints of a segment in one dimension, which causes us to lose two degrees of freedom in forming the quadrature rule, but allows us to enforce continuity between elements. In two dimensions, this translates to placing nodes on the edges of elements, and in three dimensions, placing nodes on the faces of elements. Then, if the order of the polynomial basis functions is N, then the number of GLL nodes in one dimension is $Nq = N + 1$. Taking the tensor product of the one dimensional node coordinates provides the two dimensional node coordinates. Thus, there are $Nq^2 = (N + 1)^2$ nodes in two dimensions, or on each face of a hexehedral element. Taking the tensor product again results in the

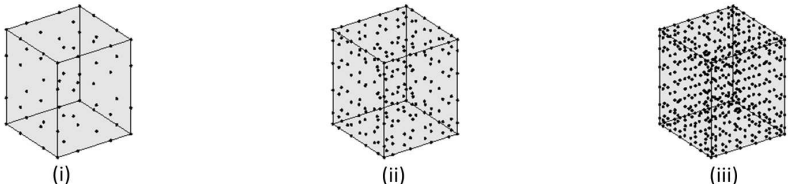
GLL nodes on a single element, where the total number of nodes is $Nq^3 = (N+1)^3$. Figure 2.1 shows the GLL nodes for different values of N in each dimension.



(a) (i) N = 3 (ii) N = 5 (iii) N = 7



(b) (i) N = 3 (ii) N = 5 (iii) N = 7



(c) (i) N = 3 (ii) N = 5 (iii) N = 7

Figure 2.1 : Gauss-Lobatto-Legendre nodes for varying N in (a) 1D, (b) 2D, and (c) 3D.

To find GLL node coordinates for each element in the domain, we begin by defining a reference element on the $(r, s, t) \in [-1, 1]^3$ coordinate system. Then, we define a set of GLL nodes on the reference element, which can be mapped to any mesh element. That is, any (r, s, t) coordinate on the reference element can be mapped to an (x, y, z)

coordinate on a mesh element. This mapping allows for faster computations of the integrals on each quadrilateral, because the integral only needs to be calculated once. That is, the integral is computed on the reference element, and then mapped to every mesh element through the transformation equations. For clarity, we first define $\omega_1(q) = 1 - q$ and $\omega_2(q) = 1 + q$. Then, the transformation equations are

$$\begin{aligned}
x = \frac{1}{8}\Big[&\omega_1(r)\omega_1(s)\omega_1(t)x_1^k + \omega_2(r)\omega_1(s)\omega_1(t)x_2^k + \omega_2(r)\omega_2(s)\omega_1(t)x_3^k + \omega_1(r)\omega_2(s)\omega_1(t)x_4^k \\
&+ \omega_1(r)\omega_1(s)\omega_2(t)x_5^k + \omega_2(r)\omega_1(s)\omega_2(t)x_6^k + \omega_2(r)\omega_2(s)\omega_2(t)x_7^k + \omega_1(r)\omega_2(s)\omega_2(t)x_8^k\Big] \\
y = \frac{1}{8}\Big[&\omega_1(r)\omega_1(s)\omega_1(t)y_1^k + \omega_2(r)\omega_1(s)\omega_1(t)y_2^k + \omega_2(r)\omega_2(s)\omega_1(t)y_3^k + \omega_1(r)\omega_2(s)\omega_1(t)y_4^k \\
&+ \omega_1(r)\omega_1(s)\omega_2(t)y_5^k + \omega_2(r)\omega_1(s)\omega_2(t)y_6^k + \omega_2(r)\omega_2(s)\omega_2(t)y_7^k + \omega_1(r)\omega_2(s)\omega_2(t)y_8^k\Big] \\
z = \frac{1}{8}\Big[&\omega_1(r)\omega_1(s)\omega_1(t)z_1^k + \omega_2(r)\omega_1(s)\omega_1(t)z_2^k + \omega_2(r)\omega_2(s)\omega_1(t)z_3^k + \omega_1(r)\omega_2(s)\omega_1(t)z_4^k \\
&+ \omega_1(r)\omega_1(s)\omega_2(t)z_5^k + \omega_2(r)\omega_1(s)\omega_2(t)z_6^k + \omega_2(r)\omega_2(s)\omega_2(t)z_7^k + \omega_1(r)\omega_2(s)\omega_2(t)z_8^k\Big],
\end{aligned} \tag{2.1}$$

where x_i^k, y_i^k, and z_i^k are the respective x, y, and z coordinates of the i^{th} vertex on the k^{th} element. This mapping is shown in Figure 2.2.

2.1.3 Node Numbering

Once Nq^3 and the node coordinates on each mesh element are known, the nodes are numbered. At first, we number the nodes in each element individually, and consequently the total number of nodes is $Nq^3 * K$. However, this original numbering system does not take into account connections between elements, and so the total number of nodes is not the number of degrees of freedom. Thus, to number the

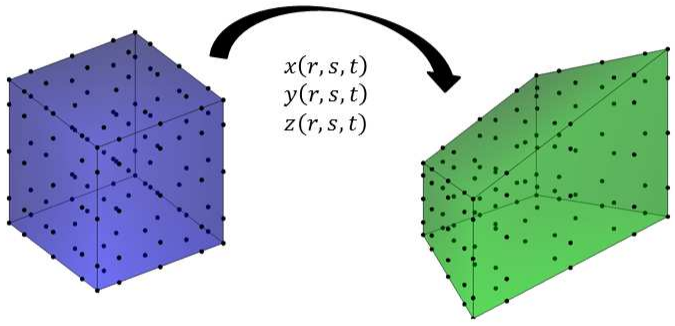
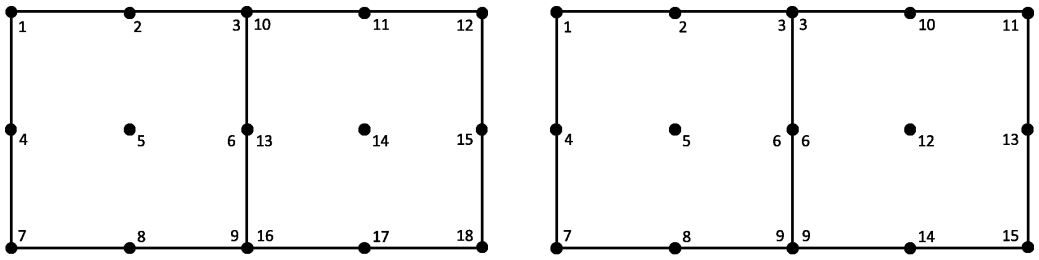


Figure 2.2 : Mapping of GLL nodes from the reference element $\{(r,s,t) \in [-1,1]^3\}$ to a mesh element.

degrees of freedom, we identify nodes that are shared by more than one element.

First, we create a matrix `galnums` $\in \mathbb{R}^{Nq^3 \times K}$, where the entries are numbered from 1 to $Nq^3 * K$. Then, comparing the coordinates of the face nodes on each element, one can find the nodes that are shared by more than one element. From this, the minimum node number between both entries in `galnums` becomes the node number in both positions of the matrix. Finally, after renumbering shared nodes in the mesh, we renumber the remaining nodes to ensure a contiguous global numbering of all nodes. This is done by ignoring the shared nodes that were changed, and contiguously numbering the remaining nodes. Now, `galnums` contains only the unique

global node numbers, or degrees of freedom, for the entire domain. Thus, `galnums` is now a mapping from local node numbers to global node numbers. Figure 2.3 shows the local and global node numbering for a two dimensional mesh with two elements, and $N = 2$.



(a) Local Node Numbers. (b) Global Node Numbers.

Figure 2.3 : Local (a) and global (b) node numbering for a two dimensional domain with two elements and $N = 2$.

2.1.4 Basis Functions

The next part of the spectral element method is the basis functions. This thesis uses Lagrangian basis functions:

$$\begin{aligned}\hat{\phi}_{ijk}(x,y,z) &= \phi_i\left(r(x,y,z)\right)\phi_j\left(s(x,y,z)\right)\phi_k\left(t(x,y,z)\right)\\ &= \phi_i(r)\phi_j(s)\phi_k(t),\end{aligned}$$

where ϕ_i, ϕ_j, and ϕ_k are the Lagrangian polynomials at the GLL nodes. In particular, the value of the basis functions at the GLL nodes can be written as

$$\begin{aligned}\hat{\phi}_{ijk}(x_{mnp},y_{mnp},z_{mnp}) &= \phi_i(r_m)\phi_j(s_n)\phi_k(t_n)\\ &= \delta_{im}\delta_{jn}\delta_{kp},\end{aligned}$$

where δ_{ij} is the Kronecker delta function:

$$\delta_{ij} = \begin{cases} 0, & \text{if } i \neq j \\ 1, & \text{if } i = j. \end{cases}$$

Thus, using the Lagrangian polynomials and the tensor product form of the GLL nodes, we represent the solution to the weak formulation of the incompressible Navier Stokes equations in Equation (1.2), u, as a tensor product.

$$u = \sum_{e=1}^{K} \sum_{m=1}^{Nq} \sum_{n=1}^{Nq} \sum_{p=1}^{Nq} u^e_{mnp} \phi^e_m(r) \phi^e_n(s) \phi^e_p(t), \tag{2.2}$$

where u^e_{mnp} is the nodal basis coefficient.

Note that the derivatives of the basis functions, with respect to x, y, and z, are

$$\begin{aligned}
\frac{\partial \hat{\phi}_{ijk}}{\partial x} &= \left(\frac{\partial \phi_i(r)}{\partial r} \phi_j(s) \phi_k(t) \right) \frac{\partial r}{\partial x} + \left(\phi_i(r) \frac{\partial \phi_j(s)}{\partial s} \phi_k(t) \right) \frac{\partial s}{\partial x} + \left(\phi_i(r) \phi_j(s) \frac{\partial \phi_k(t)}{\partial t} \right) \frac{\partial t}{\partial x} \\
\frac{\partial \hat{\phi}_{ijk}}{\partial y} &= \left(\frac{\partial \phi_i(r)}{\partial r} \phi_j(s) \phi_k(t) \right) \frac{\partial r}{\partial y} + \left(\phi_i(r) \frac{\partial \phi_j(s)}{\partial s} \phi_k(t) \right) \frac{\partial s}{\partial y} + \left(\phi_i(r) \phi_j(s) \frac{\partial \phi_k(t)}{\partial t} \right) \frac{\partial t}{\partial y} \\
\frac{\partial \hat{\phi}_{ijk}}{\partial z} &= \left(\frac{\partial \phi_i(r)}{\partial r} \phi_j(s) \phi_k(t) \right) \frac{\partial r}{\partial z} + \left(\phi_i(r) \frac{\partial \phi_j(s)}{\partial s} \phi_k(t) \right) \frac{\partial s}{\partial z} + \left(\phi_i(r) \phi_j(s) \frac{\partial \phi_k(t)}{\partial t} \right) \frac{\partial t}{\partial z}.
\end{aligned} \tag{2.3}$$

From this, one can see that the derivatives of u with respect to each direction, x, y,

and z, are

$$\begin{aligned}
\frac{\partial u}{\partial x} &= \sum_{e=1}^{K} \sum_{m,n,p=1}^{Nq} u^e_{mnp} \Bigg[\left(\frac{\partial \phi^e_m(r)}{\partial r}\, \phi^e_n(s)\, \phi^e_p(t) \right) \frac{\partial r}{\partial x} + \left(\phi^e_m(r)\, \frac{\partial \phi^e_n(s)}{\partial s}\, \phi^e_p(t) \right) \frac{\partial s}{\partial x} \\
&\qquad + \left(\phi^e_m(r)\, \phi^e_n(s)\, \frac{\partial \phi^e_p(t)}{\partial t} \right) \frac{\partial t}{\partial x} \Bigg] \\
\frac{\partial u}{\partial y} &= \sum_{e=1}^{K} \sum_{m,n,p=1}^{Nq} u^e_{mnp} \Bigg[\left(\frac{\partial \phi^e_m(r)}{\partial r}\, \phi^e_n(s)\, \phi^e_p(t) \right) \frac{\partial r}{\partial y} + \left(\phi^e_m(r)\, \frac{\partial \phi^e_n(s)}{\partial s}\, \phi^e_p(t) \right) \frac{\partial s}{\partial y} \\
&\qquad + \left(\phi^e_m(r)\, \phi^e_n(s)\, \frac{\partial \phi^e_p(t)}{\partial t} \right) \frac{\partial t}{\partial y} \Bigg] \\
\frac{\partial u}{\partial z} &= \sum_{e=1}^{K} \sum_{m,n,p=1}^{Nq} u^e_{mnp} \Bigg[\left(\frac{\partial \phi^e_m(r)}{\partial r}\, \phi^e_n(s)\, \phi^e_p(t) \right) \frac{\partial r}{\partial z} + \left(\phi^e_m(r)\, \frac{\partial \phi^e_n(s)}{\partial s}\, \phi^e_p(t) \right) \frac{\partial s}{\partial z} \\
&\qquad + \left(\phi^e_m(r)\, \phi^e_n(s)\, \frac{\partial \phi^e_p(t)}{\partial t} \right) \frac{\partial t}{\partial z} \Bigg]
\end{aligned} \tag{2.4}$$

Then, for a single element, e, at a node $(x_{abc}, y_{abc}, z_{abc})$, we write $\frac{\partial u}{\partial x}(r_a, s_b, t_c)$ as

$$\begin{aligned}
\frac{\partial u}{\partial x}(r_a, s_b, t_c) = \sum_{m,n,p=1}^{Nq} u^e_{mnp} \Bigg[& \left(\frac{\partial \phi^e_m(r_a)}{\partial r}\, \phi^e_n(s_b)\, \phi^e_p(t_c) \right) \frac{\partial r}{\partial x}(r_a, s_b, t_c) \\
& + \left(\phi^e_m(r_a)\, \frac{\partial \phi^e_n(s_b)}{\partial s}\, \phi^e_p(t_c) \right) \frac{\partial s}{\partial x}(r_a, s_b, t_c) \\
& + \left(\phi^e_m(r_a)\, \phi^e_n(s_b)\, \frac{\partial \phi^e_p(t_c)}{\partial t} \right) \frac{\partial t}{\partial x}(r_a, s_b, t_c) \Bigg].
\end{aligned} \tag{2.5}$$

Thus, as one can see, there are certain geometric factors, namely $\frac{\partial r}{\partial (x,y,z)}$, $\frac{\partial s}{\partial (x,y,z)}$, and $\frac{\partial t}{\partial (x,y,z)}$, needed to compute the partial derivatives of the basis functions, $\frac{\partial\, \hat{\phi}_{ijk}}{\partial (x,y,z)}$, and the approximate solution, $\frac{\partial u}{\partial (x,y,z)}$.

2.1.5 Geometric Factors

The geometric factors needed to compute the partial derivatives in Equations (2.3) and (2.4) are constant, because they are on the reference element. Thus, the geometric

factors are precomputed and stored in a matrix, `gfacs`. The rows of `gfacs` are made up of the partial derivatives described above, and the Jacobian that arises when using a change of variables. Specifically, when integrating on a mesh element, the integral is first computed on the reference element and then mapped to the mesh element using a change of variables. Thus, for some function $f(x, y, z)$,

$$\int_{D^k} f(x,y,z)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = \int_{\hat{D}} f\left(x(r,s,t), y(r,s,t), z(r,s,t)\right)\, J\,\mathrm{d}r\,\mathrm{d}s\,\mathrm{d}t,$$

where $x(r,s,t)$, $y(r,s,t)$, and $z(r,s,t)$ come from the transformation equations (2.1), D^k is the mesh element, and $\hat{D}$ is the reference element. Also, J is the Jacobian:

$$J = \frac{\partial(x,y,z)}{\partial(r,s,t)} = \begin{vmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} & \frac{\partial z}{\partial t} \end{vmatrix} \tag{2.6}$$

$$J = \frac{\partial x}{\partial r}\frac{\partial y}{\partial s}\frac{\partial z}{\partial t} + \frac{\partial x}{\partial s}\frac{\partial y}{\partial t}\frac{\partial z}{\partial r} + \frac{\partial x}{\partial t}\frac{\partial y}{\partial r}\frac{\partial z}{\partial s} - \frac{\partial x}{\partial t}\frac{\partial y}{\partial s}\frac{\partial z}{\partial r} - \frac{\partial x}{\partial s}\frac{\partial y}{\partial r}\frac{\partial z}{\partial t} - \frac{\partial x}{\partial r}\frac{\partial y}{\partial t}\frac{\partial z}{\partial s}.$$

Now, to find the geometric factors, observe that [27]

$$\frac{\partial x}{\partial r}\frac{\partial r}{\partial x} = \begin{pmatrix} \frac{\partial r}{\partial x} & \frac{\partial s}{\partial x} & \frac{\partial t}{\partial x} \\ \frac{\partial r}{\partial y} & \frac{\partial s}{\partial y} & \frac{\partial t}{\partial y} \\ \frac{\partial r}{\partial z} & \frac{\partial s}{\partial z} & \frac{\partial t}{\partial z} \end{pmatrix} \begin{pmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} & \frac{\partial z}{\partial t} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{2.7}$$

Using this information and Equation (2.6), we see that

$$\begin{pmatrix} \frac{\partial r}{\partial x} & \frac{\partial s}{\partial x} & \frac{\partial t}{\partial x} \\ \frac{\partial r}{\partial y} & \frac{\partial s}{\partial y} & \frac{\partial t}{\partial y} \\ \frac{\partial r}{\partial z} & \frac{\partial s}{\partial z} & \frac{\partial t}{\partial z} \end{pmatrix} = \frac{1}{J} \begin{pmatrix} \frac{\partial z}{\partial t}\frac{\partial y}{\partial s} - \frac{\partial z}{\partial s}\frac{\partial y}{\partial t} & -\left(\frac{\partial z}{\partial t}\frac{\partial x}{\partial s} - \frac{\partial z}{\partial s}\frac{\partial x}{\partial t}\right) & \frac{\partial y}{\partial t}\frac{\partial x}{\partial s} - \frac{\partial y}{\partial s}\frac{\partial x}{\partial t} \\ -\left(\frac{\partial z}{\partial t}\frac{\partial y}{\partial r} - \frac{\partial z}{\partial r}\frac{\partial y}{\partial t}\right) & \frac{\partial z}{\partial t}\frac{\partial x}{\partial r} - \frac{\partial z}{\partial r}\frac{\partial x}{\partial t} & -\left(\frac{\partial y}{\partial t}\frac{\partial x}{\partial r} - \frac{\partial y}{\partial r}\frac{\partial x}{\partial t}\right) \\ \frac{\partial z}{\partial s}\frac{\partial y}{\partial r} - \frac{\partial z}{\partial r}\frac{\partial y}{\partial s} & -\left(\frac{\partial z}{\partial s}\frac{\partial x}{\partial r} - \frac{\partial z}{\partial r}\frac{\partial x}{\partial s}\right) & \frac{\partial y}{\partial s}\frac{\partial x}{\partial r} - \frac{\partial y}{\partial r}\frac{\partial x}{\partial s} \end{pmatrix} \tag{2.8}$$

From this, the geometric factors are

$$\begin{aligned}
\frac{\partial r}{\partial x} &= \frac{\frac{\partial z}{\partial t}\frac{\partial y}{\partial s} - \frac{\partial z}{\partial s}\frac{\partial y}{\partial t}}{J}, & \frac{\partial r}{\partial y} &= \frac{-\left(\frac{\partial z}{\partial t}\frac{\partial x}{\partial s} - \frac{\partial z}{\partial s}\frac{\partial x}{\partial t}\right)}{J}, & \frac{\partial r}{\partial z} &= \frac{\frac{\partial y}{\partial t}\frac{\partial x}{\partial s} - \frac{\partial y}{\partial s}\frac{\partial x}{\partial t}}{J}, \\
\frac{\partial s}{\partial x} &= \frac{-\left(\frac{\partial z}{\partial t}\frac{\partial y}{\partial r} - \frac{\partial z}{\partial r}\frac{\partial y}{\partial t}\right)}{J}, & \frac{\partial s}{\partial y} &= \frac{\frac{\partial z}{\partial t}\frac{\partial x}{\partial r} - \frac{\partial z}{\partial r}\frac{\partial x}{\partial t}}{J}, & \frac{\partial s}{\partial z} &= \frac{-\left(\frac{\partial y}{\partial t}\frac{\partial x}{\partial r} - \frac{\partial y}{\partial r}\frac{\partial x}{\partial t}\right)}{J}, \\
\frac{\partial t}{\partial x} &= \frac{\frac{\partial z}{\partial s}\frac{\partial y}{\partial r} - \frac{\partial z}{\partial r}\frac{\partial y}{\partial s}}{J}, & \frac{\partial t}{\partial y} &= \frac{-\left(\frac{\partial z}{\partial s}\frac{\partial x}{\partial r} - \frac{\partial z}{\partial r}\frac{\partial x}{\partial s}\right)}{J}, & \frac{\partial t}{\partial z} &= \frac{\frac{\partial y}{\partial s}\frac{\partial x}{\partial r} - \frac{\partial y}{\partial r}\frac{\partial x}{\partial s}}{J},
\end{aligned} \tag{2.9}$$

where the partial derivatives $\frac{\partial(x,y,z)}{\partial(r,s,t)}$, are the derivatives of the respective transformation equations (2.1). In the interest of space, shown below are only the partial derivatives of x with respect to r, s, and t. Note that the partials of y and z are similar:

$$\begin{aligned}
\frac{\partial x}{\partial r} &= \frac{1}{8}\Big[-\omega_1(s)\omega_1(t)x_1^k + \omega_1(s)\omega_1(t)x_2^k + \omega_2(s)\omega_1(t)x_3^k - \omega_2(s)\omega_1(t)x_4^k \\
&\qquad - \omega_1(s)\omega_2(t)x_5^k + \omega_1(s)\omega_2(t)x_6^k + \omega_2(s)\omega_2(t)x_7^k - \omega_2(s)\omega_2(t)x_8^k\Big] \\
\frac{\partial x}{\partial s} &= \frac{1}{8}\Big[-\omega_1(r)\omega_1(t)x_1^k - \omega_2(r)\omega_1(t)x_2^k + \omega_2(r)\omega_1(t)x_3^k + \omega_1(r)\omega_1(t)x_4^k \\
&\qquad - \omega_1(r)\omega_2(t)x_5^k - \omega_2(r)\omega_2(t)x_6^k + \omega_2(r)\omega_2(t)x_7^k + \omega_1(r)\omega_2(t)x_8^k\Big] \\
\frac{\partial x}{\partial t} &= \frac{1}{8}\Big[-\omega_1(r)\omega_1(s)x_1^k - \omega_2(r)\omega_1(s)x_2^k - \omega_2(r)\omega_2(s)x_3^k - \omega_1(r)\omega_2(s)x_4^k \\
&\qquad + \omega_1(r)\omega_1(s)x_5^k + \omega_2(r)\omega_1(s)x_6^k + \omega_2(r)\omega_2(s)x_7^k + \omega_1(r)\omega_2(s)x_8^k\Big].
\end{aligned} \tag{2.10}$$

Thus, precomputing the geometric factors and storing them in `gfacs` saves computational time, because we simply access them in memory when computing integrals, specifically when implementing the quadrature rule.

2.1.6 Quadrature

As mentioned in Section 2.1.2, this thesis uses a Gauss-Lobatto-Legendre quadrature rule to numerically compute integrals [48]. The GLL quadrature rule uses specific

weights associated with each node to approximate an integral. That is, for a function $f(r,s,t)$, the integral is approximated on the reference element as

$$\int_{\hat{D}} f(r,s,t) \approx \sum_{m=1}^{Nq}\sum_{n=1}^{Nq}\sum_{p=1}^{Nq} w_m w_n w_p f(r_{mnp}, s_{mnp}, t_{mnp}).$$

This quadrature rule is implemented when solving the variational problem in Equation (1.2), because we must compute the energy inner product of u and the basis functions $\hat{\phi}_{ijk}$ on each element:

$$(\nabla\hat{\phi}_{ijk}, \nabla u)_{D^k} = \int_{D^k} \nabla\hat{\phi}_{ijk}(x,y,z)\,\nabla u(x,y,z)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z. \tag{2.11}$$

Again, using a change of variables, we compute the inner product in Equation (2.11) using the reference element, $\hat{D}$:

$$\int_{D^k} \nabla\hat{\phi}_{ijk}(x,y,z)\,\nabla u(x,y,z)\,\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z = \int_{\hat{D}} \nabla\hat{\phi}_{ijk}(r,s,t)\,\nabla u(r,s,t)\,J\,\mathrm{d}r\,\mathrm{d}s\,\mathrm{d}t. \tag{2.12}$$

Now, to compute the inner product in Equation (2.12), we use the associated quadrature weights and the partial derivatives in Equation (2.4). We only show the partial derivatives with respect to x, and note that the partial derivatives with respect to y and z are similar:

$$\begin{aligned}
\frac{\partial\hat{\phi}_{ijk}}{\partial x}\frac{\partial u}{\partial x} =& \left[\left(\frac{\mathrm{d}\phi_i(r)}{\mathrm{d}r}\,\phi_j(s)\,\phi_k(t)\right)\frac{\partial r}{\partial x}\right.\\
&\left.+\left(\phi_i(r)\,\frac{\mathrm{d}\phi_j(s)}{\mathrm{d}s}\,\phi_k(t)\right)\frac{\partial s}{\partial x}+\left(\phi_i(r)\,\phi_j(s)\,\frac{\mathrm{d}\phi_k(t)}{\mathrm{d}t}\right)\frac{\partial t}{\partial x}\right]\\
&\left[\sum_{e=1}^{K}\sum_{m,n,p=1}^{Nq} u^e_{mnp}\left(\left(\frac{\mathrm{d}\phi^e_m(r)}{\mathrm{d}r}\,\phi^e_n(s)\,\phi^e_p(t)\right)\frac{\partial r}{\partial x}\right.\right.\\
&\left.\left.+\left(\phi^e_m(r)\,\frac{\mathrm{d}\phi^e_n(s)}{\mathrm{d}s}\,\phi^e_p(t)\right)\frac{\partial s}{\partial x}+\left(\phi^e_m(r)\,\phi^e_n(s)\,\frac{\mathrm{d}\phi^e_p(t)}{\mathrm{d}t}\right)\frac{\partial t}{\partial x}\right)\right].
\end{aligned} \tag{2.13}$$

Then, applying the quadrature rule, we approximate the integral on an element, e, at a node $(x_{abc}, y_{abc}, z_{abc})$, by

$$\int_{D^e} \frac{\partial \hat{\phi}_{ijk}}{\partial x}\frac{\partial u}{\partial x} \approx \sum_{a,b,c=1}^{Nq} w_a w_b w_c \left[\frac{\partial \hat{\phi}_{ijk}}{\partial x}(x_{abc}, y_{abc}, z_{abc})\right]\left[\frac{\partial u}{\partial x}(x_{abc}, y_{abc}, z_{abc})\right]$$

$$\begin{aligned}
&= \sum_{a,b,c=1}^{Nq} w_a w_b w_c J^e_{abc}\Bigg[\left(\frac{\mathrm{d}\phi_i(r_a)}{\mathrm{d}r}\,\phi_j(s_b)\,\phi_k(t_c)\right)\,\frac{\partial r^e}{\partial x}(r_a, s_b, t_c) \\
&\qquad + \left(\phi_i(r_a)\,\frac{\mathrm{d}\phi_j(s_b)}{\mathrm{d}s}\,\phi_k(t_c)\right)\,\frac{\partial s^e}{\partial x}(r_a, s_b, t_c) \\
&\qquad + \left(\phi_i(r_a)\,\phi_j(s_b)\,\frac{\mathrm{d}\phi_k(t_c)}{\mathrm{d}t}\right)\,\frac{\partial t^e}{\partial x}(r_a, s_b, t_c)\Bigg] \\
&\quad \Bigg[\sum_{m,n,p=1}^{Nq} u^e_{abc}\Bigg(\left(\frac{\mathrm{d}\phi_m(r_a)}{\mathrm{d}r}\,\phi_n(s_b)\,\phi_p(t_c)\right)\,\frac{\partial r^e}{\partial x}(r_a, s_b, t_c) \\
&\qquad + \left(\phi_m(r_a)\,\frac{\mathrm{d}\phi_n(s_b)}{\mathrm{d}s}\,\phi_p(t_c)\right)\,\frac{\partial s^e}{\partial x}(r_a, s_b, t_c) \\
&\qquad + \left(\phi_m(r_a)\,\phi_n(s_b)\,\frac{\mathrm{d}\phi_p(t_c)}{\mathrm{d}t}\right)\,\frac{\partial t^e}{\partial x}(r_a, s_b, t_c)\Bigg)\Bigg].
\end{aligned} \tag{2.14}$$

Finally, using the tensor product form in Equation (2.14), we can write the following inner product as a matrix multiplication [54]:

$$\left(\nabla\hat{\phi}, \nabla u\right)_{D^e} = (\phi^e)^T \begin{pmatrix} D_r \\ D_s \\ D_t \end{pmatrix}^T \begin{pmatrix} G_{rr} & G_{rs} & G_{rt} \\ G_{sr} & G_{ss} & G_{st} \\ G_{tr} & G_{ts} & G_{tt} \end{pmatrix} \begin{pmatrix} D_r \\ D_s \\ D_t \end{pmatrix} (u^e) \tag{2.15}$$

Thus, we define the following geometric factors, which can be precomputed:

$$G_{rr} = (W \otimes W \otimes W)\ J \left(\frac{\partial r}{\partial x}\frac{\partial r}{\partial x} + \frac{\partial r}{\partial y}\frac{\partial r}{\partial y} + \frac{\partial r}{\partial z}\frac{\partial r}{\partial z} \right)$$

$$G_{rs} = G_{sr} = (W \otimes W \otimes W)\ J \left(\frac{\partial r}{\partial x}\frac{\partial s}{\partial x} + \frac{\partial r}{\partial y}\frac{\partial s}{\partial y} + \frac{\partial r}{\partial z}\frac{\partial s}{\partial z} \right)$$

$$G_{rt} = (W \otimes W \otimes W)\ J \left(\frac{\partial r}{\partial x}\frac{\partial t}{\partial x} + \frac{\partial r}{\partial y}\frac{\partial t}{\partial y} + \frac{\partial r}{\partial z}\frac{\partial t}{\partial z} \right)$$

$$G_{ss} = (W \otimes W \otimes W)\ J \left(\frac{\partial s}{\partial x}\frac{\partial s}{\partial x} + \frac{\partial s}{\partial y}\frac{\partial s}{\partial y} + \frac{\partial s}{\partial z}\frac{\partial s}{\partial z} \right)$$

$$G_{st} = G_{ts} = (W \otimes W \otimes W)\ J \left(\frac{\partial s}{\partial x}\frac{\partial t}{\partial x} + \frac{\partial s}{\partial y}\frac{\partial t}{\partial y} + \frac{\partial s}{\partial z}\frac{\partial t}{\partial z} \right)$$

$$G_{tt} = (W \otimes W \otimes W)\ J \left(\frac{\partial t}{\partial x}\frac{\partial t}{\partial x} + \frac{\partial t}{\partial y}\frac{\partial t}{\partial y} + \frac{\partial t}{\partial z}\frac{\partial t}{\partial z} \right)$$

Here $M \otimes N$ denotes the Kronecker product between matrices M and N. Also, W is the one dimensional vector of GLL weights, and J is the Jacobian. Then, the differentiation matrices are defined as

$$D_r = I \otimes I \otimes D, \quad D_s = I \otimes D \otimes I, \quad D_t = D \otimes I \otimes I.$$

The entries in D are the derivatives of the Lagrangian polynomials.

$$D_{ij} = \frac{\mathrm{d}\phi_j(r_i)}{\mathrm{d}r}$$

2.2 Splitting Method

Fully discretizing Equation (1.1) requires insertion of the spectral element bases into the weak formulation of the problem in Equation (1.2), which we represent in matrix form. The following formulation follows the work in [8] and [16], using similar notation

and symbols:

$$\begin{aligned} \nu \mathbf{L}\, u^{n+1} + \frac{1}{dt}\mathbf{W}\, u^{n+1} - \mathbf{D}^T p^{n+1} &= \mathbf{W}\, \hat{f}^{\,n+1} \\ \mathbf{D}\, \mathbf{u}^{n+1} &= 0. \end{aligned} \tag{2.16}$$

Here, $\mathbf{L}$ is the discrete Laplace operator, $\mathbf{D}$ is the discrete divergence operator, and $\mathbf{W}$ is the diagonal mass matrix whose entries are the GLL weights. Finally, the right hand side $\hat{f}^{\,n+1}$ is the explicit information for the advection term. Note that the capital bold face letters represent matrices that act on vector fields. Thus, at each time step, we solve the system in Equation (2.16).

This is done through the splitting method described in Chapter 1. Thus, the steps in Equations (1.3) - (1.5) can be written in the following form. Here, the first equation is the weak form of the splitting step, and the second equation is the matrix form.

1. Advection Step:

$$\begin{aligned} (\phi, \tilde{u})_\Omega &= \left(\phi, u^n - dt\hat{f}^{\,n+1}\right)_\Omega \\ \tilde{u} &= M_G^{-1}\widehat{\mathbf{W}}\left(u^n - dt\hat{f}^{\,n+1}\right) \end{aligned} \tag{2.17}$$

2. Pressure Step

$$\begin{aligned} \left(\phi, \tilde{\tilde{u}}\right)_\Omega &= \left(\phi, \tilde{u} - dt\,\nabla p^{n+1}\right)_\Omega \\ \tilde{\tilde{u}} &= M_G^{-1}\widehat{\mathbf{W}}\left(\tilde{u} - dt\,\nabla p^{n+1}\right) \end{aligned} \tag{2.18}$$

3. Diffusion Step:

$$\begin{aligned} \left(\phi, u^{n+1}\right)_\Omega &= \left(\phi, \frac{dt}{Re}u^{n+1} + \tilde{\tilde{u}}\right)_\Omega \\ \mathbf{A}\, u^{n+1} &= -\frac{1}{\nu dt}\widehat{\mathbf{W}}\,\tilde{\tilde{u}} \end{aligned} \tag{2.19}$$

In the Advection step (2.17), $\hat{f}^{n+1} = (u^n \cdot \nabla)\, u^n$ and

$$\widehat{\mathbf{W}} = (W \otimes W \otimes W)\, J.$$

Here, $M_G^{-1} = Q^T M_L Q$ is the Galerkin inverse mass matrix, where Q^T and Q are the respective gather and scatter operators, and M_L is the matrix of local mass matrices. We take advantage of the geometry and the definition of the GLL node sets by using the tensor product of W, which is the one dimensional vector of GLL weights; J is the Jacobian. In the Diffusion step (2.19), $\mathbf{A} = \mathbf{L} + \frac{1}{\nu dt}\mathbf{W}$.

In this section, we describe each of the splitting steps in detail, highlighting the tensor product form of Equations (2.17) - (2.19). This form is useful in the implementation of the method, which will be discussed in Chapter 3.

2.2.1 Step 1: Advection

In the first step, u^n and dt are known quantities, and we solve for the intermediate velocity $\tilde{u}$ explicitly. To do so, we first compute the advection term $\hat{f}^{n+1} = (u^n \cdot \nabla)\, u^n$, where $u^n \in \mathbb{R}^{Nq^3 * K * 3 \times 1}$ is a vector, and K is the total number of elements in the mesh. That is, $u^n = (u \quad v \quad w)^T$, where u, v, $w \in \mathbb{R}^{Nq^3 * K \times 1}$ are vectors that contain the velocity values in the x, y, and z directions, respectively. Thus, the operator $(u^n \cdot \nabla)$ can be written as

$$\begin{pmatrix} u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y} + w\frac{\partial}{\partial z} \\ u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y} + w\frac{\partial}{\partial z} \\ u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y} + w\frac{\partial}{\partial z} \end{pmatrix}. \tag{2.20}$$

Then, applying $(u^n \cdot \nabla)$ to u^n, the advection term is

$$\begin{pmatrix} u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} \\ u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} \\ u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} \end{pmatrix}. \tag{2.21}$$

Using the partial derivatives on the reference element (r, s, t) coordinate system and the chain rule, we can find the partial derivatives on the (x, y, z) coordinate system:

$$\begin{aligned} \frac{\partial u}{\partial x} &= \frac{\partial r}{\partial x}\frac{\partial u}{\partial r} + \frac{\partial s}{\partial x}\frac{\partial u}{\partial s} + \frac{\partial t}{\partial x}\frac{\partial u}{\partial t} \\ \frac{\partial u}{\partial y} &= \frac{\partial r}{\partial y}\frac{\partial u}{\partial r} + \frac{\partial s}{\partial y}\frac{\partial u}{\partial s} + \frac{\partial t}{\partial y}\frac{\partial u}{\partial t} \\ \frac{\partial u}{\partial z} &= \frac{\partial r}{\partial z}\frac{\partial u}{\partial r} + \frac{\partial s}{\partial z}\frac{\partial u}{\partial s} + \frac{\partial t}{\partial z}\frac{\partial u}{\partial t}. \end{aligned} \tag{2.22}$$

Thus, we can rewrite the term $u^n \cdot \nabla u^n$ as

$$\begin{pmatrix} \hat{u}\frac{\partial u}{\partial r} + \hat{v}\frac{\partial u}{\partial s} + \hat{w}\frac{\partial u}{\partial t} \\ \hat{u}\frac{\partial v}{\partial r} + \hat{v}\frac{\partial v}{\partial s} + \hat{w}\frac{\partial v}{\partial t} \\ \hat{u}\frac{\partial w}{\partial r} + \hat{v}\frac{\partial w}{\partial s} + \hat{w}\frac{\partial w}{\partial t} \end{pmatrix} \tag{2.23}$$

where $\hat{u}$, $\hat{v}$, and $\hat{w}$ are defined as

$$\begin{aligned} \hat{u} &= u\frac{\partial r}{\partial x} + v\frac{\partial r}{\partial y} + w\frac{\partial r}{\partial z} \\ \hat{v} &= u\frac{\partial s}{\partial x} + v\frac{\partial s}{\partial y} + w\frac{\partial s}{\partial z} \\ \hat{w} &= u\frac{\partial t}{\partial x} + v\frac{\partial t}{\partial y} + w\frac{\partial t}{\partial z}. \end{aligned} \tag{2.24}$$

Note that u, v, and w are known values. Then, the geometric factors, $\frac{\partial r}{\partial(x,y,z)}$, $\frac{\partial s}{\partial(x,y,z)}$, and $\frac{\partial t}{\partial(x,y,z)}$, were precomputed in Equation (2.9), and stored in `gfacs`. Finally, the differentiation matrix is used to compute the remaining partial derivatives of the velocity: $\frac{\partial u}{\partial(r,s,t)}$, $\frac{\partial v}{\partial(r,s,t)}$, and $\frac{\partial w}{\partial(r,s,t)}$. The tensor product form of the solution in Equation

(2.2) allows us to write these partial derivatives locally as (only one partial derivative is shown, however the others are analagous):

$$\frac{\partial u(r_a, s_b, t_c)}{\partial r} = \sum_{m=1}^{Nq} u_{mbc} \frac{\mathrm{d}\phi_m(r_a)}{\mathrm{d}r}, \tag{2.25}$$

where D is the differentation matrix whose entries are the derivatives of the Lagrangian polynomials:

$$D_{ij} = \frac{\mathrm{d}\phi_j(r_i)}{\mathrm{d}r}. \tag{2.26}$$

To finish computing $\tilde{u}$ in the Advection step of Equation (2.17), one must multiply $(u^n \cdot \nabla)\, u^n$ by dt and add u^n. Then, to implement the quadrature and compute the integral, we incorporate the mass matrix, $\widehat{\mathbf{W}}$, to get

$$\tilde{u} = \widehat{\mathbf{W}} \left(u^n + dt \hat{f}^{\,n+1} \right).$$

2.2.2 Step 2: Pressure

The Pressure step incorporates the pressure into the solution. This requires a substep, where we first compute the pressure at the current time step. Then the intermediate velocity, $\tilde{\tilde{u}}$, is found using the gradient of the computed pressure value. Thus, we first take the divergence of the Step 2 equation (1.4):

$$\nabla \cdot \left(\frac{\tilde{\tilde{u}} - \tilde{u}}{dt} \right) = \nabla \cdot (-\nabla p^{n+1}).$$

Then, following the assumptions made by Karniadakis et al. [30], we seek p^{n+1} such that $\nabla \cdot \tilde{\tilde{u}} = 0$, and the above equation becomes

$$\frac{\nabla \cdot \tilde{u}}{dt} = \Delta p^{n+1}. \tag{2.27}$$

Thus, the first part of the pressure step is to compute the divergence of $\tilde{u}$,

$$\nabla \cdot \tilde{u} = \frac{\partial \tilde{u}}{\partial x} + \frac{\partial \tilde{v}}{\partial y} + \frac{\partial \tilde{w}}{\partial z},$$

where the partial derivatives can be rewritten using the chain rule, as in Equation (2.22),

$$\begin{aligned}
\frac{\partial \tilde{u}}{\partial x} &= \frac{\partial r}{\partial x}\frac{\partial \tilde{u}}{\partial r} + \frac{\partial s}{\partial x}\frac{\partial \tilde{u}}{\partial s} + \frac{\partial t}{\partial x}\frac{\partial \tilde{u}}{\partial t} \\
\frac{\partial \tilde{v}}{\partial y} &= \frac{\partial r}{\partial y}\frac{\partial \tilde{v}}{\partial r} + \frac{\partial s}{\partial y}\frac{\partial \tilde{v}}{\partial s} + \frac{\partial t}{\partial y}\frac{\partial \tilde{v}}{\partial t} \\
\frac{\partial \tilde{w}}{\partial z} &= \frac{\partial r}{\partial z}\frac{\partial \tilde{w}}{\partial r} + \frac{\partial s}{\partial z}\frac{\partial \tilde{w}}{\partial s} + \frac{\partial t}{\partial z}\frac{\partial \tilde{w}}{\partial t}.
\end{aligned}$$

Thus, the divergence of $\tilde{u}$ is equal to the following summation:

$$\begin{aligned}
&\nabla \cdot \tilde{u} = \\
&\frac{\partial r}{\partial x}\frac{\partial \tilde{u}}{\partial r} + \frac{\partial s}{\partial x}\frac{\partial \tilde{u}}{\partial s} + \frac{\partial t}{\partial x}\frac{\partial \tilde{u}}{\partial t} + \frac{\partial r}{\partial y}\frac{\partial \tilde{v}}{\partial r} + \frac{\partial s}{\partial y}\frac{\partial \tilde{v}}{\partial s} + \frac{\partial t}{\partial y}\frac{\partial \tilde{v}}{\partial t} + \frac{\partial r}{\partial z}\frac{\partial \tilde{w}}{\partial r} + \frac{\partial s}{\partial z}\frac{\partial \tilde{w}}{\partial s} + \frac{\partial t}{\partial z}\frac{\partial \tilde{w}}{\partial t},
\end{aligned} \tag{2.28}$$

where the partial derivates, $\frac{\partial \tilde{u}}{\partial(r,s,t)}$, $\frac{\partial \tilde{v}}{\partial(r,s,t)}$, and $\frac{\partial \tilde{w}}{\partial(r,s,t)}$, are analagous to that of Equation (2.25), and the geometric factors are stored in `gfacs`.

The next part of the Pressure step is to solve the screened Coulomb equation (2.27), with $\lambda = 0$, for the pressure. To solve this equation, we solve the equivalent weak formulation, where we take the inner product of Equation (2.27) with a Lagrangian test function ϕ, and use Green's Theorem to get the formulation

$$\left(\phi, \frac{\nabla \cdot \tilde{u}}{dt}\right)_{\Omega} = \left(\phi, \frac{\partial p^{n+1}}{\partial n}\right)_{\partial\Omega} + \left(\nabla\phi, \nabla p^{n+1}\right)_{\Omega}. \tag{2.29}$$

The $(\phi, \frac{\partial p^{n+1}}{\partial n})_{\partial\Omega}$ term in Equation (2.29) requires knowledge of the pressure Neumann boundary conditions, for which this thesis uses the boundary conditions proposed by Orszag et al. [40]. That is, the pressure Neumann boundary conditions are

derived by evaluating the momentum equation normal to the boundary:

$$n \cdot \left(\frac{\partial u}{\partial t} + (u \cdot \nabla)\, u = -\nabla p + \nu \Delta u \right) \Big|_{\partial\Omega}. \tag{2.30}$$

Assuming velocity is zero on the boundary walls, or that there is a no slip boundary condition for the velocity, implies that the time derivative term is zero:

$$\frac{\partial u}{\partial t}\Big|_{\partial\Omega} = 0.$$

We also make use of the following identity to rewrite the diffusion term, Δu:

$$\nabla \times (\nabla \times\ u) = \nabla(\nabla \cdot u) - \Delta u$$

$$\Rightarrow \quad \Delta u = \nabla(\nabla \cdot u) - \nabla \times (\nabla \times\ u).$$

Then, using the assumption on velocity and the previous identity, Equation (2.30) becomes

$$n \cdot \nabla p = n \cdot \left(-(u \cdot \nabla)\, u + \frac{1}{Re} \nu\, (\Delta u) \right)$$

$$\Rightarrow \quad \frac{\partial p}{\partial n}\Big|_{\partial\Omega} = n \cdot \Big(-(u \cdot \nabla)\, u + \nu \big[\nabla(\nabla \cdot u) - \nabla \times (\nabla \times\ u) \big] \Big).$$

Finally, imposing the incompressibility constraint, $\nabla \cdot u = 0$, implies that

$$\frac{\partial p}{\partial n}\Big|_{\partial\Omega} = -\, n \cdot \Big((u \cdot \nabla)\, u + \nu \big[\nabla \times (\nabla \times\ u) \big] \Big). \tag{2.31}$$

Thus, to solve the screened Coulomb potential equation in (2.27) for the pressure, we use the derived Neumann boundary condition in Equation (2.31) in the preconditioned conjugate gradient method discussed in Section 2.3.

Then, to find the second intermediate velocity, $\tilde{\tilde{u}}$, we find the gradient of the computed pressure. This process again utilizes Equation (2.22), where we use the

chain rule and geometric factors to represent the partial derivatives of the pressure:

$$\begin{aligned}
\frac{\partial p}{\partial x} &= \frac{\partial r}{\partial x}\frac{\partial p}{\partial r} + \frac{\partial s}{\partial x}\frac{\partial p}{\partial s} + \frac{\partial t}{\partial x}\frac{\partial p}{\partial t} \\
\frac{\partial p}{\partial y} &= \frac{\partial r}{\partial y}\frac{\partial p}{\partial r} + \frac{\partial s}{\partial y}\frac{\partial p}{\partial s} + \frac{\partial t}{\partial y}\frac{\partial p}{\partial t} \\
\frac{\partial p}{\partial z} &= \frac{\partial r}{\partial z}\frac{\partial p}{\partial r} + \frac{\partial s}{\partial z}\frac{\partial p}{\partial s} + \frac{\partial t}{\partial z}\frac{\partial p}{\partial t}.
\end{aligned}$$

Then, taking advantage of the tensor product form, the terms $\frac{\partial p}{\partial (r,s,t)}$ can be written in the same form as Equation (2.25), for which we use the differentiation matrix, D:

$$\frac{\partial p\left(r_a, s_b, t_c\right)}{\partial r} = \sum_{m=1}^{Nq} p_{mbc}\frac{\partial \phi_m(r_a)}{\partial r} = D\, p. \tag{2.32}$$

Thus,

$$\nabla p^{n+1} = \begin{pmatrix} \frac{\partial p}{\partial x} & \frac{\partial p}{\partial y} & \frac{\partial p}{\partial z} \end{pmatrix}^T \tag{2.33}$$

After computing ∇p^{n+1}, we multiply it by dt and then subtract that value from the $\tilde{u}$ found in the Advection step. Then, again to incorporate the quadrature rule and compute the integral, we use the matrix $\widehat{\mathbf{W}}$:

$$\tilde{\tilde{u}} = \widehat{\mathbf{W}}\left(\tilde{u} - dt\nabla p^{n+1}\right).$$

2.2.3 Step 3: Diffusion

In the final step, or the diffusion step, the diffusion term from Equation (1.1a) is incoporated to find the velocity, u^{n+1}:

$$u^{n+1} = \tilde{\tilde{u}} + dt\,\nu\,\Delta u^{n+1}.$$

Thus, we solve the screened Coulomb potential problem

$$\begin{aligned} u^{n+1} - dt\nu\Delta u^{n+1} &= \tilde{\tilde{u}} \\ \Rightarrow \quad \Delta u^{n+1} - \frac{1}{dt\nu}u^{n+1} &= \frac{-1}{dt\nu}\tilde{\tilde{u}} \end{aligned} \tag{2.34}$$

for u^{n+1}, where $\lambda = \frac{1}{dt\nu}$. The forcing function is $f = \frac{-1}{dt\nu}\tilde{\tilde{u}}$, where $\tilde{\tilde{u}}$ was computed in the Pressure step. Again, we solve the weak formulation of Equation (2.34):

$$-\left(\nabla\phi, \nabla u^{n+1}\right)_\Omega + \left(\phi, \frac{u^{n+1}}{dt\nu}\right)_\Omega = \left(\phi, \frac{-\tilde{\tilde{u}}}{dt\nu}\right)_\Omega - \left(\phi, \frac{\partial u^{n+1}}{\partial n}\right)_{\partial\Omega}, \tag{2.35}$$

for a test function ϕ. Then, using the spectral element bases in Equation (2.35), we want to solve the system in Equation (2.19).

$$\mathbf{A}\, u^{n+1} = -\frac{1}{\nu dt}\widehat{\mathbf{W}}\,\tilde{\tilde{u}}$$

Further, as stated in Section 2.2.2, we have no slip boundary conditions for the velocity, $u|_{\partial\Omega} = 0$. Thus, $\left(\phi, \frac{\partial u^{n+1}}{\partial n}\right)_{\partial\Omega} = 0$. Hence, the Diffusion step reduces to solving Equation (2.35) for u^{n+1}, using Dirichlet velocity boundary conditions, and the preconditioned conjugate gradient method discussed in Section 2.3.

2.3 Screened Coulomb Potential Problem

Both the Pressure and Diffusion steps require a solution of the screened Coulomb potential (SCP) problem, which requires us to solve the PDE

$$(\Delta - \lambda)u = f, \tag{2.36}$$

where λ is a constant. When solving this PDE, we actually solve the weak formulation. That is, we take the inner product of each term with the test function, ϕ,

$$(\Delta u, \phi)_\Omega + \lambda(u, \phi)_\Omega = (f, \phi)_\Omega$$

and then use Green's Theorem to obtain the variational form:

$$(\nabla u, \nabla \phi)_\Omega - \left(\frac{\partial u}{\partial n}, \phi\right)_{\partial\Omega} + \lambda(u, \phi)_\Omega = (f, \phi)_\Omega. \tag{2.37}$$

Similar to Equation (1.2), we can represent Equation (2.37) using the spectral element bases. That is, we can write the weak form of the SCP problem in matrix form:

$$\mathbf{A}u = \mathbf{W}f,$$

where $\mathbf{W}$ is again the diagonal mass matrix with the GLL weights on the diagonal, and $\mathbf{A} = \mathbf{L} - \lambda\mathbf{W}$, with $\mathbf{L}$ as the discrete Laplace operator.

2.3.1 Preconditioned Conjugate Gradient Method

In the SCP problem of Equation (2.36), we want to solve a system of the following form.

$$\mathbf{A}u = f, \quad \text{where } \mathbf{A} = A_r + A_s + A_t$$

A_r, A_s, and A_t are the one dimensional stiffness matrices. Then, given that A_r, A_s, and A_t are diagonalizable, we can write this system as [35]

$$\left(S_r^{-1} A_r S_r + S_s^{-1} A_s S_s + S_t^{-1} A_t S_t\right) u = f.$$

Note that this formulation neglects the cross terms of the Laplacian. In other words, we neglect the terms that involve combinations of the (r, s, t) derivatives. Thus, we only use the diagonal terms of the correct Laplacian matrix. Solving this system using the PCG method requires inversion of $\mathbf{A}$ at each iteration. Because $\mathbf{A}$ is both diagonalizable and tensorizable, we can construct the inverse of $\mathbf{A}$ [35] as

$$\mathbf{A}^{-1} = S_r \otimes S_s \otimes S_t \left(I \otimes I \otimes \Lambda_r + I \otimes \Lambda_s \otimes I + \Lambda_t \otimes I \otimes I\right) S_r^{-1} \otimes S_s^{-1} \otimes S_t^{-1}, \quad (2.38)$$

where Λ_r, Λ_s, and Λ_t are diagonal matrices of eigenvalues that satisfy the one dimensional eigenvalue equation $\mathbf{A}u = \lambda u$, and S_r, S_s, and S_t are matrices whose columns are the corresponding eigenvectors [35].

In Chapter 1, we refered to the additive overlapping Schwarz predonditioning explored in [38] and [10]. This technique is based on domain decomposition where the local subdomains are discretized using finite elements and GLL nodes in each subdomain [13], [17], [52]. This method uses finite element-based preconditioners, where as using the fast diagonalization method and tensor product forms we can consider a restriction of the matrix $\mathbf{A}$ that arises from the spectral element method [6].

Thus, we first define the subdomains, $\bar{D}^e$, used to set up the preconditioner. For each element, $\bar{D}^e$ is the subdomain that extends to one GLL node past the boundary of D^e. Thus, each $\bar{D}^e$ is simply the original element D^e extended by one plane of nodal values. An example of these subdomains is shown in Figure 2.4. Thus, we define the restriction matrix R_e, that selects the nodal values for $\bar{D}^e$ from the global stiffness matrix [16].

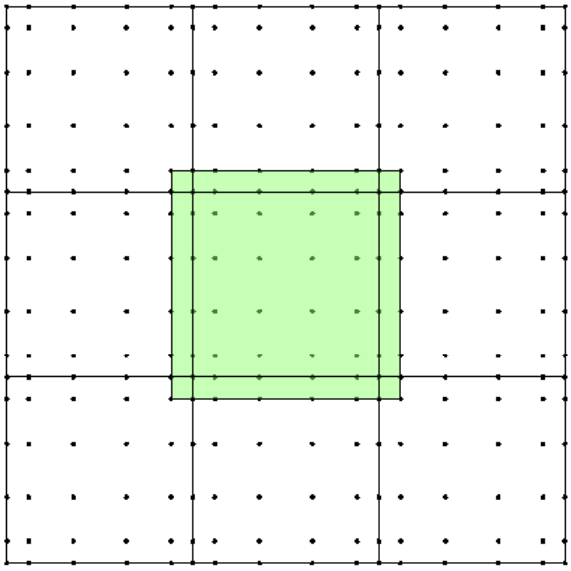


Figure 2.4 : Pictured is a single layer overlapping subdomain for a two dimensional domain with $K = 9$, $N = 5$.

When setting up the preconditioner, we configure $\bar{D}^e$ in each direction separately. In other words, we get the one dimensional stiffness and mass matrices A_r and W_r. Then, using the one dimensional restriction matrix R_e, we extract the appropriate nodal values from both A_r and W_r. We then obtain the eigenvectors, S_r and eigenvalues Λ_r by solving the following eigenvalue problem

$$\left(R_e^T\, A_r\, R_e\right) S_r = \left(R_e^T\, W_r\, R_e\right) S_r\, \Lambda_r. \tag{2.39}$$

Note that configuring $A_s\ A_t$ and $W_s\ W_t$ are similar. Thus, applying this restriction to

Equation (2.38), we see that solving the local problems requires the following inverse,

$$(\mathbf{A}^e)^{-1} = \tilde{S}_r \otimes \tilde{S}_s \otimes \tilde{S}_t \left(I \otimes I \otimes \tilde{\Lambda}_r + I \otimes \tilde{\Lambda}_s \otimes I + \tilde{\Lambda}_t \otimes I \otimes I \right) \tilde{S}_r^{-1} \otimes \tilde{S}_s^{-1} \otimes \tilde{S}_t^{-1}, \tag{2.40}$$

where the pairs $(\tilde{S}_r, \tilde{\Lambda}_r)$, $(\tilde{S}_s, \tilde{\Lambda}_s)$, and $(\tilde{S}_t, \tilde{\Lambda}_t)$ solve the eigenvalue problem in Equation (2.39).

Thus, the preconditioner applies this inverse to the right hand side vector f to solve the approximate local problems on each of the subdomains $\bar{D}^e$. Then, we use a gather-scatter operation to get the influence from the neighboring elements, and the resulting transformed right hand side vector $\tilde{f}$ becomes the initial guess for the conjugate gradient method. The details behind the implementation of this preconditioner are discussed in Section 3.5.

2.3.2 Constant Correction

In the preconditioned conjugate gradient method, we search for the best approximation to the solution of an equation of the form $Ax = b$, where A is a matrix, and x and b are both vectors. The solution x, to this problem need not be unique for a singular matrix A. In fact, we can shift x by a constant $c \in Ker(A)$, such that $\hat{x} = x + c$ is still a solution to $Ax = b$.

The issue of uniqueness in the solution, x, is eliminated when we enforce a Dirichlet condition on the boundary. That is, we know the value of x on the Dirichlet boundary: $x = g \quad \text{on} \quad \partial\Omega_D$, which must be satisfied by all solutions. Thus, all solutions must

satisfy $\hat{x} = x + c = g \text{on} \partial\Omega_D$, which in effect, fixes the constant and hence the uniqueness of the solution.

In the instance where we have all Neumann boundary conditions, we do not specify the value of the solution on the boundary, rather the normal derivative of the solution on the boundary: $\frac{\partial x}{\partial n} = h \quad \text{on} \quad \partial\Omega_N$. Thus, any $\hat{x} = x + c$ is a solution to the system of equations. At each iteration in the conjugate gradient method, we introduce this constant, where we represent the solution vector as $\hat{x} = x + c\vec{1}$, where $\vec{1}$ is a vector of all ones. This constant, namely $\vec{1}$, is in the null space of A, which means the system $Ax = b$ is a singular system.

In order to account for this constant, and alleviate this problem, we can shift the spectrum of A before we go through a conjugate gradient iteration. That is, before each iteration, we adjust Ax to be $\hat{A}x = \left(A + \vec{1} \cdot \vec{1}^T\right) x$. Then, the new system is $\hat{A}x = Ax + \vec{1} \cdot \vec{1}^T x = b$. Thus, we shift each entry of Ax by the sum of the entries in x:

$$\hat{A} = Ax + sum(x)$$

This ensures that the system $\hat{A}x = b$ is not singular, and thus has a unique solution.

We explored a different way of correcting this constant. In particular, we correct the solution vector at each iteration. That is, we subtract the projection of the solution vector onto the constant from the current solution vector. Thus, we find x, where

$$x = \hat{x} - \Pi_{\vec{1}}\, x = \hat{x} - \frac{x \cdot \vec{1}}{\vec{1} \cdot \vec{1}} \cdot \vec{1}.$$

This reduces to finding

$$x = \hat{x} - \vec{q}, \text{ where } \quad q_i = \frac{sum(x)}{n} \quad \text{for} \quad i = 1, ..., n.$$

Thus, adding this constant correction for the pressure ensures that we find a unique solution to the system $Ax = b$. Thus, at each time step, we have the method for computing each intermediate velocity for the splitting steps, and arriving at the next velocity, u^{n+1}. The following chapter then explains the details in the implementation of this spectral element solution of the incompressible Navier Stokes equations, including code explanations.

Chapter 3

Implementation

We implement this splitting scheme using OpenCL kernels that utilize the Graphics Processing Units' (GPUs) architecture to do a majority of the computations in parallel. That is, the solver uses both the CPU and GPU, sending data to the GPU as needed for computations in the kernels. We begin by showing the main header files in gNek, with a list of the functions in each file. Further, Figure 3.1 gives the dependency tree from left to right. In other words, `ins3d.hpp` depends on `scp3d.hpp`, which depends on `sem3d.hpp`, and so on. As one can see, we first get the mesh information, which reads in the element and boundary information discussed in Section 2.1.1. Then, `sem3d.hpp` creates the spectral element geometry information and numbering, and `scp3d.hpp` sets up the preconditioned conjugate gradient method to solve the screened Coulomb potential problems in the splitting scheme. Finally, `ins3d.hpp` executes the solve function, which in turn executes each of the splitting steps in Equations (1.3) - (1.5) at every time step.

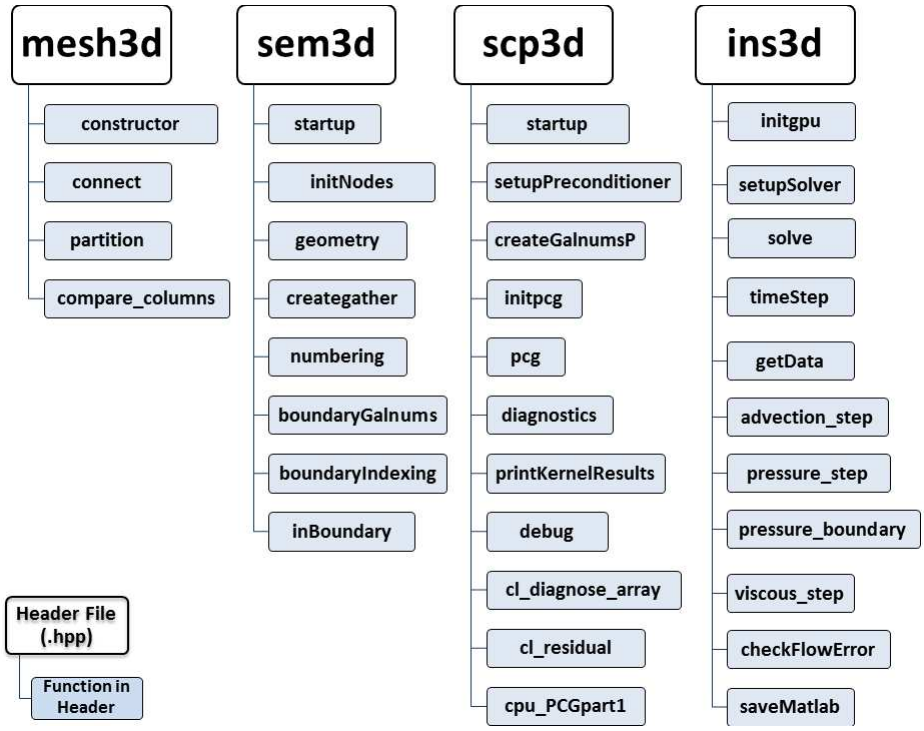


Figure 3.1 : A list of the main header files and their associated functions.

Next, we give a quick overview of the kernels used to solve the incompressible Navier Stokes equations (1.1). In Figure 3.2, we list each of the kernels called while solving Equation (1.1), with an explanation the computation performed by each kernel, and an example of the code that performs the computations.

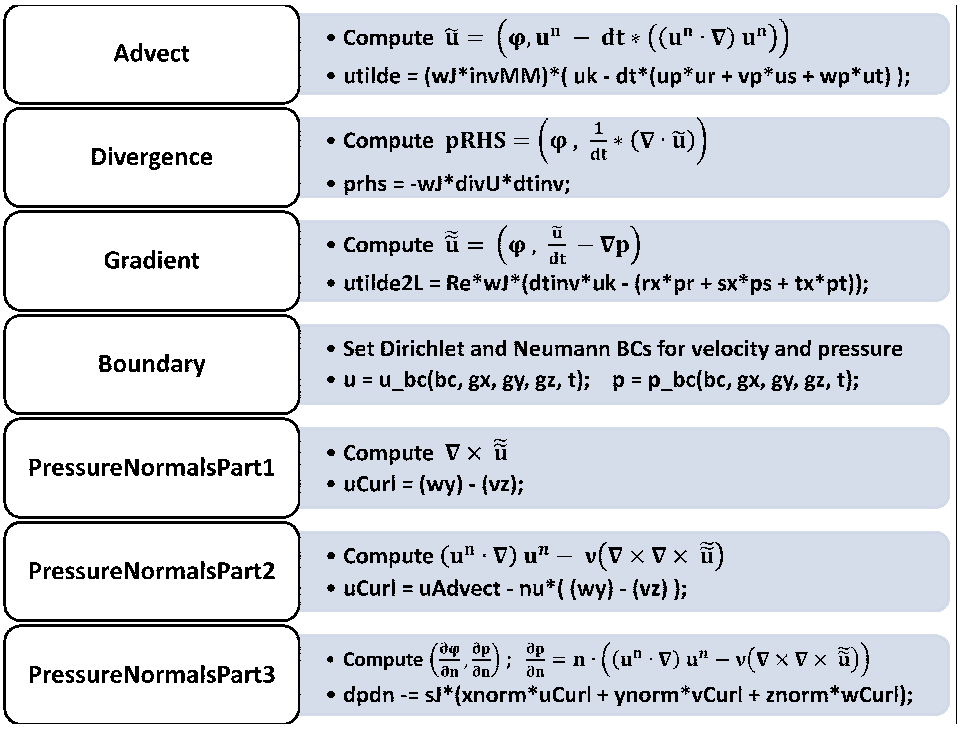


Figure 3.2 : Kernels used in the INS solve. Listed are the equations for the computation followed by the code used to perform the computation.

3.1 Data Movement

Most of the kernels in gNek utilize the memory architecture of the GPUs in the same way. Thus, this section describes the distribution of data across the global, local, and private memory in each kernel. We begin by providing a diagram of the scope of each of the memory types available in a kernel. That is, a kernel has global memory, which can be accessed by all work groups and work items. Then, the local memory in each work group can only be accessed by the work items of that work group. Finally, each work item has access to its own private memory, or register. This is illustrated

in Figure 3.3.

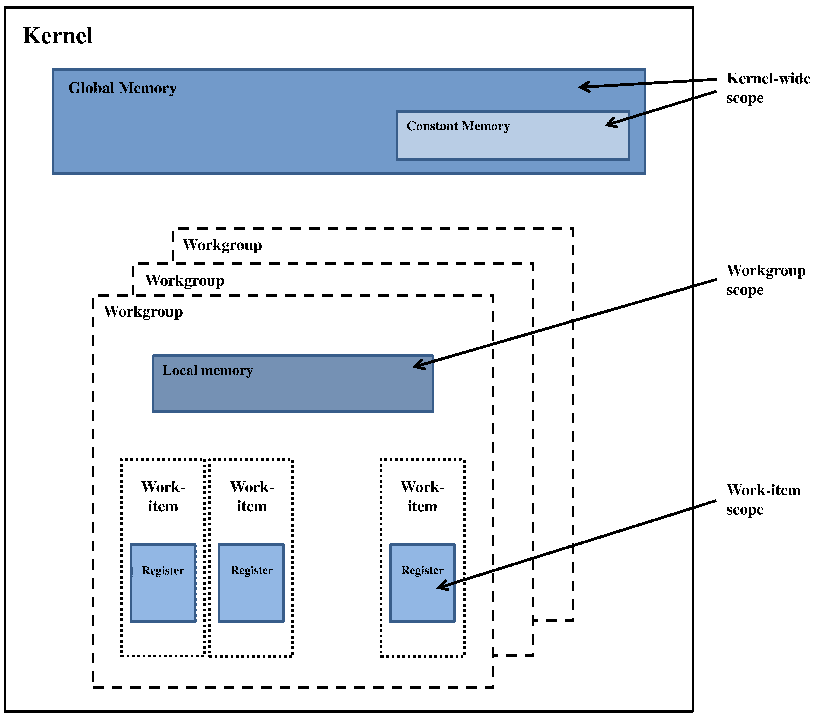


Figure 3.3 : Memory architecture in a kernel (diagram adapted from [20]).

The `Advect.cl`, `Divergence.cl`, `PressureNormalsPart1.cl`, `PressureNormalsPart2.cl`, and `Gradient.cl` kernels all take in velocity vectors u, v, and w as kernel arguments (for some kernels, this could be $\tilde{u}$ or p). The differentiation matrix, `D`, is also a common parameter, as well as `gfacs`. Thus, in each kernel we first allocate local memory for local copies of the differentiation matrix and each component of the velocity, `LD`, `Lu`, `Lv`, and `Lw`. We also allocate `uk`, `vk`, and `wk` on the registers.

```
1 /* shared register for 'r,s' plane */
2 __local myfloat LD[Nq][Nq+PAD];
3 __local myfloat Lu[Nq][Nq+PAD];
4 __local myfloat Lv[Nq][Nq+PAD];
```

```
__local myfloat Lw[Nq][Nq+PAD];

// u[:][j][i] -> uk[:]
myfloat uk[Nq]; // lots of registers...
myfloat vk[Nq];
myfloat wk[Nq];
```

In order to assign each work group to an element in the domain we set the element index, `e`, to be the group ID. Then we use the work items to perform computations for each GLL node. Thus, we set the node indicies, `i` and `j`, using the local IDs.

```
const unsigned int e = get_group_id(0);
const unsigned int i = get_local_id(0);
const unsigned int j = get_local_id(1);
unsigned int k;
```

After allocating the space, we load the differentiation matrix into local memory.

```
/* load D into local memory */
LD[i][j] = D[Nq*j+i]; // D is column major
```

When loading the respective velocity values, in order to ensure that the reads from memory are coalesced, we divide an element into Nq planes that lie on the (r, s) plane. Then, we load the velocity values into the register, where each (i, j) entry on a single plane corresponds to an entry in `uk`, `vk`, or `wk`. An illustration of the (r, s) planes and registers can be found in Figure 3.4.

```
/* *** Change load here to use scatter array *** */
/* load pencil of u into register */
unsigned int id = e*BSIZE+j*Nq+i;

for(k=0;k<Nq;++k) {
  uk[k] = u[id];
  vk[k] = v[id];
  wk[k] = w[id];
  id += Nq*Nq;
}
```

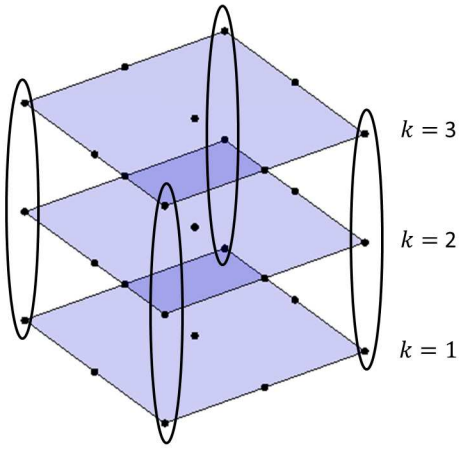


Figure 3.4 : A reference element with $Nq = 3$. Circled are four example registers, where each highlighted plane lies on the (r, s) coordinate plane. In reality, there are 9 registers total, or Nq^2.

Now, we load the velocity values into local memory, by looping through each of the (r, s) planes, where k is the "plane" ID. Thus, for each node at the (i, j) position on the k^{th} plane, we load the velocity values into the local copy of `Lu`, `Lv`, or `Lw`.

```
1  for(k=0;k<Nq;++k){
2    barrier(CLK_LOCAL_MEM_FENCE);
3
4    // share velocity from my node pencil
5    Lu[j][i] = uk[k];
6    Lv[j][i] = vk[k];
7    Lw[j][i] = wk[k];
```

In the same manner, and in the same k loop, we get the values for the geometric factors, quadrature weights, Jacobian scaling constants, and the inverse mass matrix weights at each node.

```
1  for(k=0;k<Nq;++k){
2    // prefetch geometric factors
```

```
3    id = e*11*BSIZE+k*Nq*Nq+j*Nq+i;
4    const myfloat rx = gfacs[id]; id+= BSIZE;
5    const myfloat ry = gfacs[id]; id+= BSIZE;
6    const myfloat rz = gfacs[id]; id+= BSIZE;
7
8    const myfloat sx = gfacs[id]; id+= BSIZE;
9    const myfloat sy = gfacs[id]; id+= BSIZE;
10   const myfloat sz = gfacs[id]; id+= BSIZE;
11
12   const myfloat tx = gfacs[id]; id+= BSIZE;
13   const myfloat ty = gfacs[id]; id+= BSIZE;
14   const myfloat tz = gfacs[id]; id+= BSIZE;
15
16   const myfloat wJ = gfacs[id]; id+= BSIZE;
17   const myfloat invMM = gfacs[id];
18
19   barrier(CLK_LOCAL_MEM_FENCE);
```

After obtaining and properly storing the necessary information, we need to compute the derivative of each component of the velocity with respect to the reference element: $\frac{\partial u}{\partial(r,s,t)}$, $\frac{\partial v}{\partial(r,s,t)}$, and $\frac{\partial w}{\partial(r,s,t)}$. In finding these partial derivatives, at each node, we use the differentiation matrix D. In other words, we use the fact that

$$\begin{aligned}
\left(\frac{\partial u}{\partial r}\right)_{ijke} &= D_{im}u^e_{mjk}, \quad \left(\frac{\partial u}{\partial s}\right)_{ijke} = D_{jm}u^e_{imk}, \quad \left(\frac{\partial u}{\partial t}\right)_{ijke} = D_{km}u^e_{ijm} \\
\left(\frac{\partial v}{\partial r}\right)_{ijke} &= D_{im}v^e_{mjk}, \quad \left(\frac{\partial v}{\partial s}\right)_{ijke} = D_{jm}v^e_{imk}, \quad \left(\frac{\partial v}{\partial r}\right)_{ijke} = D_{km}v^e_{ijm} \qquad (3.1) \\
\left(\frac{\partial w}{\partial r}\right)_{ijke} &= D_{im}w^e_{mjk}, \quad \left(\frac{\partial w}{\partial s}\right)_{ijke} = D_{jm}w^e_{imk}, \quad \left(\frac{\partial w}{\partial r}\right)_{ijke} = D_{km}w^e_{ijm}
\end{aligned}$$

where i denotes the index in the r direction, j in the s direction, k in the t direction, e is the index indicating the element number, and m is the index for the node number. We do this using the local copy of the differentiation matrix.

```
1    myfloat ur = 0.f, us = 0.f, ut = 0.f;
2    for(int m=0;m<Nq;++m) ut += LD[k][m]*uk[m];
3    for(int m=0;m<Nq;++m) ur += LD[i][m]*Lu[j][m];
```

```
4     for(int m=0;m<Nq;++m) us += LD[j][m]*Lu[m][i];
5
6     myfloat vr = 0.f, vs = 0.f, vt = 0.f;
7     for(int m=0;m<Nq;++m) vt += LD[k][m]*vk[m];
8     for(int m=0;m<Nq;++m) vr += LD[i][m]*Lv[j][m];
9     for(int m=0;m<Nq;++m) vs += LD[j][m]*Lv[m][i];
10
11    myfloat wr = 0.f, ws = 0.f, wt = 0.f;
12    for(int m=0;m<Nq;++m) wt += LD[k][m]*wk[m];
13    for(int m=0;m<Nq;++m) wr += LD[i][m]*Lw[j][m];
14    for(int m=0;m<Nq;++m) ws += LD[j][m]*Lw[m][i];
```

In several of the kernels discussed in the following sections, we use this method for allocating and storing the velocity information, differentiation matrix, and geometric factors. Further, the partial derivatives, $\frac{\partial u}{\partial(r,s,t)}$, $\frac{\partial v}{\partial(r,s,t)}$, and $\frac{\partial w}{\partial(r,s,t)}$, are all computed in the same way. Thus, we will not discuss this data movement in the following sections, but instead refer the reader back to the above discussion when needed.

3.2 Advection Step

3.2.1 Advection Kernel

The `Advect.cl` kernel is a two dimensional kernel that takes the geometric factors, `gfacs`, differentiation matrix `D`, u, v, w, $\tilde{u}$, $\tilde{v}$, and $\tilde{w}$ as kernel arguments. Here, $\tilde{u}$ is a pointer to the vector for the intermediate velocity to be computed in the `Advect.cl` kernel.

```
1  __kernel void Advect(const myfloat dt,
2                       __global const myfloat * restrict gfacs,
3                       __global const myfloat * restrict D,
4                       __global const myfloat * restrict u,
5                       __global const myfloat * restrict v,
6                       __global const myfloat * restrict w,
```

```
7                           __global myfloat * restrict utilde,
8                           __global myfloat * restrict vtilde,
9                           __global myfloat * restrict wtilde){
```

Then, using this information and the work groups and work items on the GPU, the `Advect.cl` kernel returns the value of the inner product at each node of each element. These values can then be added together to get the following inner product,

$$\begin{aligned} \tilde{u} &= (\phi, u^n - dt(u^n \cdot \nabla)u^n)_\Omega \\ &= \widehat{\mathbf{W}}(i,j,k)\left(u^n - dt(u^n \cdot \nabla)u^n\right), \end{aligned} \tag{3.2}$$

where $\widehat{\mathbf{W}}$ is the mass matrix discussed in Section 2.2, whose entries incorporate the quadrature weights and transformation Jacobian.

As discussed in Section 3.1, we allocate the proper local and private memory for the velocity and differentiation matrix. We also fetch the geometric factors, and compute the partial derivatves $\frac{\partial(u,v,w)}{\partial(r,s,t)}$. Then we compute the intermediate values `up`, `vp`, and `wp` by adding the products of the geometric factors with the velocity components. These intermediate values correspond to $\hat{u}$, $\hat{v}$, and $\hat{w}$ in Equation (2.24) of Section 2.2:

$$\begin{aligned} \hat{u} &= u\frac{\partial r}{\partial x} + v\frac{\partial r}{\partial y} + w\frac{\partial r}{\partial z} \\ \hat{v} &= u\frac{\partial s}{\partial x} + v\frac{\partial s}{\partial y} + w\frac{\partial s}{\partial z} \\ \hat{w} &= u\frac{\partial t}{\partial x} + v\frac{\partial t}{\partial y} + w\frac{\partial t}{\partial z}. \end{aligned}$$

```
1    const myfloat up = rx*uk[k] + ry*vk[k] + rz*wk[k];
2    const myfloat vp = sx*uk[k] + sy*vk[k] + sz*wk[k];
3    const myfloat wp = tx*uk[k] + ty*vk[k] + tz*wk[k];
```

Finally, we use `up`, `vp`, and `wp` to compute the intermediate velocity components $\tilde{u}$, $\tilde{v}$, and $\tilde{w}$. From Section 2.2, the components of the advection term, $(u^n \cdot \nabla)\, u^n$, can be written as

$$\begin{aligned}(u \cdot \nabla)\, u &= \hat{u}\frac{\partial u}{\partial r} + \hat{v}\frac{\partial u}{\partial s} + \hat{w}\frac{\partial u}{\partial t} \\ (v \cdot \nabla)\, v &= \hat{u}\frac{\partial v}{\partial r} + \hat{v}\frac{\partial v}{\partial s} + \hat{w}\frac{\partial v}{\partial t} \\ (w \cdot \nabla)\, w &= \hat{u}\frac{\partial w}{\partial r} + \hat{v}\frac{\partial w}{\partial s} + \hat{w}\frac{\partial w}{\partial t}.\end{aligned}$$

This final computation must also evaluate the inner product in Equation (3.2) at each node:

$$\tilde{u}^e_{ijk} = \left(\phi, u^n - dt(u^n \cdot \nabla)u^n\right)_{ijke},$$

which we approximate using the quadrature rule. That is, for each (m, n, p) quadrature node on the element, we have

$$\begin{aligned}\tilde{u}^e_{ijk} &\approx w_i w_j w_k J^e_{ijk} \left(u^n - dt(u^n \cdot \nabla)u^n\right) \\ &= \widehat{\mathbf{W}} \left(u^n - dt(u^n \cdot \nabla)u^n\right).\end{aligned} \tag{3.3}$$

In the implementation, the quadrature and Jacobian weights are built into the matrix $\widehat{\mathbf{W}}$, where the entry for each node was prefetched from `gfacs`, and stored in `wJ` (refer to Section 3.1). We also ensure the velocity is ready to gather, by including the mass matrix inverse weight, `invMM`.

```
1    // scaled for mass matrix inverse * pre-gather *
2    id = e*BSIZE+k*Nq*Nq+j*Nq+i;
3    utilde[id] = (wJ*invMM)*( uk[k] - dt*(up*ur+vp*us+wp*ut) );
4    vtilde[id] = (wJ*invMM)*( vk[k] - dt*(up*vr+vp*vs+wp*vt) );
5    wtilde[id] = (wJ*invMM)*( wk[k] - dt*(up*wr+vp*ws+wp*wt) );
6  }//end of k loop
```

Thus, each work item in a work group on the GPU has computed the integral in Equation (3.3) at a single node for a single element. Then, after summing these values over all elements and nodes in the `GatherScatter.cl` kernel, we will have the desired inner product in Equation (3.2):

$$(\phi, \tilde{u}) = (\phi, u^n - dt(u^n \cdot \nabla)u^n)_\Omega \,.$$

3.2.2 GatherScatter Kernel

The `GatherScatter.cl` kernel performs the gather operation followed by the scatter operation. Thus, we will explain this kernel and omit explanations for the `Gather.cl` and `Scatter.cl` kernels. On one hand, the gather operation sums the local entries in a vector that correspond to a singe global node. On the other hand, the scatter operation copies each global entry in a vector to the corresponding local entries. The `GatherScatter.cl` kernel arguments are N, `starts`, `indices`, and u.

```
1 __kernel void GatherScatter(const int N,
2                              __global int * restrict starts,
3                              __global int * restrict indices,
4                              __global myfloat *u){
```

In order to perform the gather and scatter operations, we utilize `starts` and `indices`. The vector `starts` contains the offsets for the global nodes, or degrees of freedom. The vector `indices` contains the indices of the local nodes. Thus, the list of indices, ID_{x_n} for all local nodes that correspond to the n^{th} global node are represented as

$$ID_{x_n} = [\texttt{indices}(\texttt{starts}(n)), ..., \texttt{indices}(\texttt{starts}(n+1))] \,. \tag{3.4}$$

```
1   // find coordinate of thread in global axis-0
2   int n = get_global_id(0);
3
4   if(n>=N) return;
5
6   int start = starts[n];      // coalesced
7   int end   = starts[n+1];    // coalesced ?
8
9   myfloat gun = 0;
```

In the kernel, we implement the gather operation in a for loop. For the n^{th} global node, we simply add the corresponding local entries in u, which come from Equation (3.4). Thus, the resulting sum is the continuous solution for the n^{th} global node, gun.

```
1   for(int m=start;m<end;++m){
2     const int ind = indices[m]; // not coalesced
3     if(ind>=0)
4     gun += u[ind];
5   }
```

After gathering the local entries that correspond to a single global node, we copy this continuous solution back to each entry in u. Again, we use a for loop to execute this scatter operation.

```
1   for(int m=start;m<end;++m){
2     const int ind = indices[m]; // not coalesced
3     if(ind>=0)
4     u[ind] = gun;
5   }
```

Thus, after performing both the gather and scatter operation at each node in u, we have the continuous solution for each local node in the vector, and can then perform the next operation, such as computing the divergence of u.

3.3 Pressure Step

3.3.1 Divergence Kernel

The `Divergence.cl` kernel is a two dimensional kernel that takes $\frac{1}{dt}$, `gfacs`, `D`, $\tilde{u}$, $\tilde{v}$, $\tilde{w}$, and `prhs` as kernel arguments. Here, `prhs` is the forcing term, or right hand side, for the pressure Poisson problem of Equation (2.27) that arises in the pressure step.

```
__kernel void Divergence(const myfloat dtinv,
                         __global const myfloat * restrict gfacs,
                         __global const myfloat * restrict D,
                         __global const myfloat * restrict utilde,
                         __global const myfloat * restrict vtilde,
                         __global const myfloat * restrict wtilde,
                         __global myfloat * restrict prhs){
```

The kernel then uses the work items on the GPU to compute

$$\texttt{prhs}^{e}_{ijk} = \left(\phi, \frac{\nabla \cdot \tilde{u}}{dt}\right)_{ijke}$$

at each node.

We again treat the velocity components, differentiation matrix, geometric factors, quadrature and Jacobian weights as in Section 3.1. However, loading each component of the velocity into registers, in the `Divergence.cl` kernel, we are using $\tilde{u}$, $\tilde{v}$, and $\tilde{w}$.

```
  // Change load here to use scatter array
  // Load pencil of u into register
  unsigned int id = e*BSIZE+j*Nq+i;

  for(k=0;k<Nq;++k) {
    uk[k] = utilde[id];
    vk[k] = vtilde[id];
    wk[k] = wtilde[id];
    id += Nq*Nq;
  }
```

After allocating and designating the memory for the inputs, we begin to build the divergence result. As discussed in Section 2.2, the divergence of $\tilde{u}$ can be written as

$$\nabla \cdot \tilde{u} = \frac{\partial \tilde{u}}{\partial x} + \frac{\partial \tilde{v}}{\partial y} + \frac{\partial \tilde{w}}{\partial z}.$$

Thus, we must find the divergence of each velocity component and then sum the results. In the `Divergence.cl` kernel, this is done one component at a time. That is, first we use the differentiation matrix to compute $\frac{\partial \tilde{u}}{\partial(r,s,t)}$.

$$\left(\frac{\partial \tilde{u}}{\partial r}\right)_{ijke} = D_{im}\tilde{u}_{mjke}, \quad \left(\frac{\partial \tilde{u}}{\partial s}\right)_{ijke} = D_{jm}\tilde{u}_{imke}, \quad \left(\frac{\partial \tilde{u}}{\partial t}\right)_{ijke} = D_{km}\tilde{u}_{ijme}$$

Then, we use the geometric factors to compute

$$\frac{\partial \tilde{u}}{\partial x} = \frac{\partial r}{\partial x}\frac{\partial \tilde{u}}{\partial r} + \frac{\partial s}{\partial x}\frac{\partial \tilde{u}}{\partial s} + \frac{\partial t}{\partial x}\frac{\partial \tilde{u}}{\partial t},$$

and add the result to the divergence value.

```
1     myfloat divU = 0;
2     id = e*BSIZE+k*Nq*Nq+j*Nq+i;
3     barrier(CLK_LOCAL_MEM_FENCE);
4
5     {
6       myfloat ur = 0.f, us = 0.f, ut = 0.f;
7
8       for(int m=0;m<Nq;++m) ut += LD[k][m]*uk[m];
9       for(int m=0;m<Nq;++m) ur += LD[i][m]*Lu[j][m];
10      for(int m=0;m<Nq;++m) us += LD[j][m]*Lu[m][i];
11
12      divU += rx*ur + sx*us + tx*ut;
13    }
```

Similarly, we compute $\frac{\partial \tilde{v}}{\partial(r,s,t)}$,

$$\left(\frac{\partial \tilde{v}}{\partial r}\right)_{ijke} = D_{im}\tilde{v}_{mjke}, \quad \left(\frac{\partial \tilde{v}}{\partial s}\right)_{ijke} = D_{jm}\tilde{v}_{imke}, \quad \left(\frac{\partial \tilde{v}}{\partial r}\right)_{ijke} = D_{km}\tilde{v}_{ijme}$$

and then use the geometric factors to find

$$\frac{\partial \tilde{v}}{\partial y} = \frac{\partial r}{\partial y}\frac{\partial \tilde{v}}{\partial r} + \frac{\partial s}{\partial y}\frac{\partial \tilde{v}}{\partial s} + \frac{\partial t}{\partial y}\frac{\partial \tilde{v}}{\partial t},$$

which we add to the divergence.

```
1     {
2       myfloat vr = 0.f, vs = 0.f, vt = 0.f;
3
4       for(int m=0;m<Nq;++m) vt += LD[k][m]*vk[m];
5       for(int m=0;m<Nq;++m) vr += LD[i][m]*Lv[j][m];
6       for(int m=0;m<Nq;++m) vs += LD[j][m]*Lv[m][i];
7
8       divU += ry*vr + sy*vs + ty*vt;
9     }
```

Finally, we perform the same computations for $\tilde{w}$:

$$\left(\frac{\partial w}{\partial r}\right)_{ijke} = D_{im}w^e_{mjk}, \quad \left(\frac{\partial w}{\partial s}\right)_{ijke} = D_{jm}w^e_{imk}, \quad \left(\frac{\partial w}{\partial r}\right)_{ijke} = D_{km}w^e_{ijm},$$

$$\text{and} \quad \frac{\partial \tilde{w}}{\partial z} = \frac{\partial r}{\partial z}\frac{\partial \tilde{w}}{\partial r} + \frac{\partial s}{\partial z}\frac{\partial \tilde{w}}{\partial s} + \frac{\partial t}{\partial z}\frac{\partial \tilde{w}}{\partial t},$$

```
1     {
2       myfloat wr = 0.f, ws = 0.f, wt = 0.f;
3
4       for(int m=0;m<Nq;++m) wt += LD[k][m]*wk[m];
5       for(int m=0;m<Nq;++m) wr += LD[i][m]*Lw[j][m];
6       for(int m=0;m<Nq;++m) ws += LD[j][m]*Lw[m][i];
7
8       divU += rz*wr + sz*ws + tz*wt;
9     }
```

which, when added to `divU`, gives the final value for the divergence of the velocity at each node, which corresponds to Equation (2.28) in Section 2.2.2.

$$\nabla \cdot \tilde{u} =$$

$$\frac{\partial r}{\partial x}\frac{\partial \tilde{u}}{\partial r} + \frac{\partial s}{\partial x}\frac{\partial \tilde{u}}{\partial s} + \frac{\partial t}{\partial x}\frac{\partial \tilde{u}}{\partial t} + \frac{\partial r}{\partial y}\frac{\partial \tilde{v}}{\partial r} + \frac{\partial s}{\partial y}\frac{\partial \tilde{v}}{\partial s} + \frac{\partial t}{\partial y}\frac{\partial \tilde{v}}{\partial t} + \frac{\partial r}{\partial z}\frac{\partial \tilde{w}}{\partial r} + \frac{\partial s}{\partial z}\frac{\partial \tilde{w}}{\partial s} + \frac{\partial t}{\partial z}\frac{\partial \tilde{w}}{\partial t}.$$

The last step is to compute the following inner product at each node:

$$\texttt{prhs}^e_{ijk} = \left(\phi, \frac{\nabla \cdot \tilde{u}}{dt}\right)_{ijke}.$$

In computing this value, we again us the GLL quadrature rule, where we scale by the quadrature weights and Jacobian:

$$\begin{aligned}\texttt{prhs} &\approx w_i w_j w_k J^e_{ijk} \left(\nabla \cdot \tilde{u}\right) \frac{1}{dt} \\ &= \widehat{\mathbf{W}}(i,j,k)(\nabla \cdot \tilde{u}) \frac{1}{dt},\end{aligned}$$

where $\widehat{\mathbf{W}}(i,j,k) = w_i w_j w_k J^e_{ijk} = \texttt{wJ}$.

```
1  // Weights build into geometric factors
2  prhs[id] = -wJ*divU*dtinv;
```

Thus, we return the right hand side value, or forcing term, needed to solve the screened Coulomb potential problem for the pressure.

As mentioned in Section 2.2, solving the screened Coulomb potential problem for the pressure reduces to solving a Poisson problem. When solving this Poisson problem, we must incorporate the derived Neumann pressure boundary conditions in Equation (2.31). This is done in three steps. First, we compute the advection and curl of the velocity, $(u \cdot \nabla) u$ and $\nabla \times u$. Second, we find the curl of the first curl result and scale it with the viscosity, to get $\nu (\nabla \times \nabla \times u)$. Finally, we extract results from the second step at the pressure Neumann faces and compute

$$\left(\phi, \frac{\partial p}{\partial n}\right)_{\partial\Omega} = \texttt{sJ} * \left[n \cdot \left((u \cdot \nabla) u + \nu (\nabla \times \nabla \times u) \right)\right]$$

at the boundary nodes, where sJ is the surface Jacobian (scaled with the quadrature weights) used in mapping the surface integral from the reference element to a mesh

element. We then increment the local right hand side pressure with these extracted values. These three steps are performed in the kernels `PressureNormalsPart1.cl`, `PressureNormalsPart2.cl`, and `PressureNormalsPart3.cl`, respectively. After executing these three kernels, we implement the preconditioned conjugate gradient method, which is discussed in Section 2.3.1.

3.3.2 PressureNormalsPart1 Kernel

The `PressureNormalsPart1.cl` kernel is again a two dimensional kernel, which takes dt, `gfacs`, `D`, u, v, and w as input pointers for the advection and curl values to be computed.

```
1  __kernel void PressureNormalsPart1(const myfloat dt,
2                       __global const myfloat * restrict gfacs,
3                       __global const myfloat * restrict D,
4                       __global const myfloat * restrict u,
5                       __global const myfloat * restrict v,
6                       __global const myfloat * restrict w,
7                       __global myfloat * restrict uAdvect,
8                       __global myfloat * restrict vAdvect,
9                       __global myfloat * restrict wAdvect,
10                      __global myfloat * restrict uCurl,
11                      __global myfloat * restrict vCurl,
12                      __global myfloat * restrict wCurl){
```

Then, this kernel executes the first part described above in finding the normal values of the pressure. That is, the `PressureNormalsPart1.cl` kernel computes the advection and the curl of the given velocity,

$$-(u \cdot \nabla)\, u \quad \text{and} \quad \nabla \times u,$$

at each GLL node. This is done using the same method as in Section 3.1 for allocating and populating registers and local memory for the differentiation matrix, each component of the velocity, and the geometric factors. The partial derivatives $\frac{\partial(u,v,w)}{\partial(r,s,t)}$ are also computed as in Section 3.1. Again, we omit the code and refer the reader to Section 3.1.

As in the `Advect.cl` kernel, we compute the intermediate values $\hat{u}$, $\hat{v}$, and $\hat{w}$ from Equation (2.24) of Section 2.2, which we call `up`, `vp`, and `wp` in the code.

$$\hat{u} = u\frac{\partial r}{\partial x} + v\frac{\partial r}{\partial y} + w\frac{\partial r}{\partial z}$$
$$\hat{v} = u\frac{\partial s}{\partial x} + v\frac{\partial s}{\partial y} + w\frac{\partial s}{\partial z}$$
$$\hat{w} = u\frac{\partial t}{\partial x} + v\frac{\partial t}{\partial y} + w\frac{\partial t}{\partial z}.$$

```
1 const myfloat up = rx*uk[k] + ry*vk[k] + rz*wk[k];
2 const myfloat vp = sx*uk[k] + sy*vk[k] + sz*wk[k];
3 const myfloat wp = tx*uk[k] + ty*vk[k] + tz*wk[k];
```

Then, these intermediate values are used to find the advection at each node:

$$-(u\cdot\nabla)\,u = -\left(\hat{u}\frac{\partial u}{\partial r} + \hat{v}\frac{\partial u}{\partial s} + \hat{w}\frac{\partial u}{\partial t}\right)$$
$$-(v\cdot\nabla)\,v = -\left(\hat{u}\frac{\partial v}{\partial r} + \hat{v}\frac{\partial v}{\partial s} + \hat{w}\frac{\partial v}{\partial t}\right)$$
$$-(w\cdot\nabla)\,w = -\left(\hat{u}\frac{\partial w}{\partial r} + \hat{v}\frac{\partial w}{\partial s} + \hat{w}\frac{\partial w}{\partial t}\right).$$

```
1 for(k=0;k<Nq;++k){
2
3   id = e*BSIZE+k*Nq*Nq+j*Nq+i;
4
5   uAdvect[id] = -(up*ur + vp*us + wp*ut);
6   vAdvect[id] = -(up*vr + vp*vs + wp*vt);
7   wAdvect[id] = -(up*wr + vp*ws + wp*wt);
```

Finally, in this kernel, we find the curl of the velocity at each node. This involves computing the partial derivatives, $\frac{\partial u}{\partial(y,z)}$, $\frac{\partial v}{\partial(x,z)}$, and $\frac{\partial w}{\partial(x,y)}$. These computations are analogous to those in Equation (2.22). Thus, to find the curl of the velocity in the x direction we first compute

$$\frac{\partial w}{\partial y} = \frac{\partial r}{\partial y}\frac{\partial w}{\partial r} + \frac{\partial s}{\partial y}\frac{\partial w}{\partial s} + \frac{\partial t}{\partial y}\frac{\partial w}{\partial t}$$
$$\frac{\partial v}{\partial z} = \frac{\partial r}{\partial z}\frac{\partial v}{\partial r} + \frac{\partial s}{\partial z}\frac{\partial v}{\partial s} + \frac{\partial t}{\partial z}\frac{\partial v}{\partial t}.$$

Then, we use these values to compute

$$\nabla \times u = \frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}.$$

```
1 uCurl[id] = (wr*ry + ws*sy + wt*ty)-(vr*rz + vs*sz + vt*tz);
```

Next, to find the curl of the velocity in the y direction, we first compute

$$\frac{\partial u}{\partial z} = \frac{\partial r}{\partial z}\frac{\partial u}{\partial r} + \frac{\partial s}{\partial z}\frac{\partial u}{\partial s} + \frac{\partial t}{\partial z}\frac{\partial u}{\partial t}$$
$$\frac{\partial w}{\partial x} = \frac{\partial r}{\partial x}\frac{\partial w}{\partial r} + \frac{\partial s}{\partial x}\frac{\partial w}{\partial s} + \frac{\partial t}{\partial x}\frac{\partial w}{\partial t},$$

which we use to find

$$\nabla \times v = \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}.$$

```
1 vCurl[id] = (ur*rz + us*sz + ut*tz)-(wr*rx + ws*sx + wt*tx);
```

Finally, to find the curl of the velocity in the z direction, we compute

$$\frac{\partial v}{\partial x} = \frac{\partial r}{\partial x}\frac{\partial v}{\partial r} + \frac{\partial s}{\partial x}\frac{\partial v}{\partial s} + \frac{\partial t}{\partial x}\frac{\partial v}{\partial t}$$
$$\frac{\partial u}{\partial y} = \frac{\partial r}{\partial y}\frac{\partial u}{\partial r} + \frac{\partial s}{\partial y}\frac{\partial u}{\partial s} + \frac{\partial t}{\partial y}\frac{\partial u}{\partial t},$$

which is used to find

$$\nabla \times w = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$$

```
wCurl[id] = (vr*rx + vs*sx + vt*tx)-(ur*ry + us*sy + ut*ty);
```

Thus, the `PressureNormalsPart1.cl` kernel returns the value at each node of the advection and curl of the velocity,

$$-(u \cdot \nabla)\, u \quad \text{and} \quad \nabla \times u,$$

namely `uAdvect`, `vAdvect`, `wAdvect`, `uCurl`, `vCurl`, and `wCurl`.

3.3.3 PressureNormalsPart2 Kernel

In the `PressureNormalsPart2.cl` kernel, we execute the second part of finding the normal values for the pressure. This kernel takes dt, `Reynolds`, `gfacs`, `D`, `uAdvect`, `vAdvect`, `wAdvect`, `uCurl`, `vCurl`, and `wCurl` as kernel arguments.

```
__kernel void PressureNormalsPart2(const myfloat dt,
                   const myfloat Reynolds,
                   __global const myfloat * restrict gfacs,
                   __global const myfloat * restrict D,
                   __global const myfloat * restrict uAdvect,
                   __global const myfloat * restrict vAdvect,
                   __global const myfloat * restrict wAdvect,
                   __global myfloat * restrict uCurl,
                   __global myfloat * restrict vCurl,
                   __global myfloat * restrict wCurl){
```

Then, using the curl input values, which were computed in the `PressureNormalsPart1.cl` kernel, we compute

$$\nu\Big(\nabla \times (\nabla \times u)\Big)$$

at each GLL node, where $\nu = \frac{1}{Re}$.

```
   myfloat nu = 1/Reynolds;
```

As in most of the kernels, we allocate and populate the components of the curl, differentiatoin matrix, geometric factors, and weights. In this kernel, we actually load the curl values into the registers.

```
1   unsigned int id = e*BSIZE+j*Nq+i;
2
3   for(k=0;k<Nq;++k) {
4     uk[k] = uCurl[id];
5     vk[k] = vCurl[id];
6     wk[k] = wCurl[id];
7     id += Nq*Nq;
8   }
```

Again, we refer the reader to Section 3.1 for the memory allocation and population code.

Next, we perform similar operations as in the `PressureNormalsPart1.cl` kernel. However, because we loaded `uCurl`, `vCurl`, and `wCurl` into the registers and local memory, we are performing these operations on the curl. That is, we first compute the partial derivatives of the curl,

$$\frac{\partial(\nabla \times w)}{\partial y}, \quad \frac{\partial(\nabla \times v)}{\partial z}, \quad \frac{\partial(\nabla \times u)}{\partial z}, \quad \frac{\partial(\nabla \times w)}{\partial x}, \quad \frac{\partial(\nabla \times v)}{\partial x}, \text{ and } \quad \frac{\partial(\nabla \times u)}{\partial y}.$$

```
1   for(k=0;k<Nq;++k){
2     barrier(CLK_LOCAL_MEM_FENCE);
3
4     myfloat ur = 0.f, us = 0.f, ut = 0.f;
5
6     for(int m=0;m<Nq;++m) ut += LD[k][m]*uk[m];
7     for(int m=0;m<Nq;++m) ur += LD[i][m]*Lu[j][m];
8     for(int m=0;m<Nq;++m) us += LD[j][m]*Lu[m][i];
9
10    myfloat vr = 0.f, vs = 0.f, vt = 0.f;
11
12    for(int m=0;m<Nq;++m) vt += LD[k][m]*vk[m];
13    for(int m=0;m<Nq;++m) vr += LD[i][m]*Lv[j][m];
```

```
14      for(int m=0;m<Nq;++m) vs += LD[j][m]*Lv[m][i];
15
16      myfloat wr = 0.f, ws = 0.f, wt = 0.f;
17
18      for(int m=0;m<Nq;++m) wt += LD[k][m]*wk[m];
19      for(int m=0;m<Nq;++m) wr += LD[i][m]*Lw[j][m];
20      for(int m=0;m<Nq;++m) ws += LD[j][m]*Lw[m][i];
```

Finally, we scale each component of the $\nabla \times (\nabla \times u)$ terms with the viscosity ν, and add the advection term found in `PressureNormalsPart1.cl`.

```
1  id = e*BSIZE+k*Nq*Nq+j*Nq+i;
2
3  /// scale curl-curl with viscosity
4  uCurl[id]=uAdvect[id]
5              - nu*((wr*ry+ws*sy+wt*ty)-(vr*rz+vs*sz+vt*tz));
6  vCurl[id]=vAdvect[id]
7              - nu*((ur*rz+us*sz+ut*tz)-(wr*rx+ws*sx+wt*tx));
8  wCurl[id]=wAdvect[id]
9              - nu*((vr*rx+vs*sx+vt*tx)-(ur*ry+us*sy+ut*ty));
```

Thus, the `PressureNormalsPart2.cl` kernel returns the value for each component of

$$\frac{\partial p}{\partial n} = -\left(u \cdot \nabla\right) u - \nu\big[\nabla \times (\nabla \times u)\big] \tag{3.5}$$

at each GLL node.

3.3.4 PressureNormalsPart3 Kernel

In the last part of computing the derived Neumann pressure boundary conditions, the `PressureNormalsPart3.cl` kernel computes the value at each Neumann pressure boundary node of the inner product:

$$\left(\phi, \frac{\partial p}{\partial n}\right)_{\partial\Omega}.$$

The `PressureNormalsPart3.cl` kernel takes the surface Jacobian, normal values, `bcType`, `Fmask`, and the final curl values computed in `PressureNormalsPart2.cl` (Equation (3.5)), and adds the value of the inner product above to the nodes that lie on the Neumann pressure boundary.

```
__kernel void
  PressureNormalsPart3(__global const myfloat * restrict sJ,
                       __global const myfloat * restrict xnorms,
                       __global const myfloat * restrict ynorms,
                       __global const myfloat * restrict znorms,
                       __global const int * restrict bcType,
                       __global const int * restrict Fmask,
                       __global myfloat * restrict uCurl,
                       __global myfloat * restrict vCurl,
                       __global myfloat * restrict wCurl,
                       __global myfloat * restrict dpdn){
```

As usual, we assign each element to a work group, and each GLL node to a work item using the group and local IDs, respectively.

```
  const unsigned int e = get_group_id(0);
  const unsigned int i = get_local_id(0);
  const unsigned int j = get_local_id(1);
```

Then, we loop through the nodes on each face of each element. In this loop, we first check `bcType`, which has a nonzero value in the position of the boundary nodes. We also check to see if the node is on the Neumann pressure boundary, using `pNeumannBC`, which returns either a true or false value.

```
  for(int f=0;f<6;++f){
    int bc = bcType[e*6+f];

    if(bc & pNeumannBC){
```

Then, if the current node is on the Neumann pressure boundary, we use `Fmask` to find

the local node number, `fm_id`, and then the global node number, `gid`.

```
        int fid   = i + Nq*j + f*Nq*Nq;
        int fm_id = Fmask[fid] - 1;
        int gid   = e*BSIZE + fm_id;

        fid += e*6*Nq2;
```

Next, we compute the value in Equation (2.31) at each of the Neumann pressure boundary nodes. That is, we first compute the dot product of the normal vector with the value computed in `PressureNormalsPart2.cl`:

$$\frac{\partial p}{\partial n} = n \cdot \hat{\hat{u}},$$

at each pressure Neumann boundary node, where

$$n = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix}, \quad \text{and} \quad \hat{\hat{u}} = \begin{pmatrix} (u \cdot \nabla)\, u + \nu\, (\nabla \times (\nabla \times u)) \\ (v \cdot \nabla)\, v + \nu\, (\nabla \times (\nabla \times v)) \\ (w \cdot \nabla)\, w + \nu\, (\nabla \times (\nabla \times w)) \end{pmatrix} = \begin{pmatrix} \texttt{uCurl} \\ \texttt{vCurl} \\ \texttt{wCurl} \end{pmatrix}.$$

That is, we compute the following sum.

$$\begin{aligned} \frac{\partial p}{\partial n} &= n_x \Big[(u \cdot \nabla)\, u + \nu\, (\nabla \times (\nabla \times u)) \Big] + n_y \Big[(v \cdot \nabla)\, v + \nu\, (\nabla \times (\nabla \times v)) \Big] \\ &\quad + n_z \Big[(w \cdot \nabla)\, w + \nu\, (\nabla \times (\nabla \times w)) \Big] \\ &= n_x\, \texttt{uCurl} + n_y\, \texttt{vCurl} + n_z\, \texttt{wCurl} \end{aligned}$$

```
    myfloat res =
      xnorms[fid]*uCurl[gid] +
      ynorms[fid]*vCurl[gid] +
      znorms[fid]*wCurl[gid];

    // Edges/Corners will eventually overlap
    // so the barrier will prevent overwrites
    barrier(CLK_GLOBAL_MEM_FENCE);
```

Now, to compute the final pressure boundary value, we use the GLL quadrature rule to compute the following inner product at each (i, j, k) node:

$$\left(\phi, \frac{\partial p}{\partial n}\right)_{\partial\Omega_{ijk}} \approx w_i\, w_j\, w_k\, sJ^e_{ijk}\, \frac{\partial p}{\partial n}\Big|_{\partial\Omega}.$$

Again, the quadrature weights are built into the surface Jacobian, $\texttt{sJ} = w_i\, w_j\, w_k\, sJ^e_{ijk}$.

```
1    // Weights built in sJ
2    dpdn[gid] -= sJ[fid]*res;
```

Thus, the `PressureNormalsPart3.cl` adds the derived Neumann pressure boundary conditions to the forcing term in Equation (2.29) at the correct boundary nodes:

$$\texttt{dpdn} = \left(\phi, \frac{\nabla \cdot \tilde{u}}{dt}\right)_{\Omega} + \left(\phi, \frac{\partial p}{\partial n}\right)_{\partial\Omega}.$$

3.3.5 Gradient Kernel

After solving the Poisson problem for the pressure, we must take the gradient of the pressure, scale it by dt, and add it to the current value of $\tilde{u}$, which is done in the `Gradient.cl` kernel. This kernel takes $\texttt{dtinv} = \frac{1}{dt}$, `Re`, `gfacs`, `D`, `pL`, $\tilde{u}$, $\tilde{v}$, $\tilde{w}$ and $\tilde{\tilde{u}}$, $\tilde{\tilde{v}}$, and $\tilde{\tilde{w}}$ as kernel arguments. Here, `pL` is the pressure value found in solving the Poisson problem above.

```
1  __kernel void Gradient(const myfloat dtinv, const myfloat Re,
2                         __global const myfloat * restrict gfacs,
3                         __global const myfloat * restrict D,
4                         __global const myfloat * restrict pL,
5                         __global const myfloat * restrict utildeL,
6                         __global const myfloat * restrict vtildeL,
7                         __global const myfloat * restrict wtildeL,
8                         __global myfloat * restrict utilde2L,
9                         __global myfloat * restrict vtilde2L,
10                        __global myfloat * restrict wtilde2L){
```

Similar to the previous kernels (see Section 3.1), we set the appropriate memory for the velocity, differentiation matrix, quadrature weights, and geometric factors. In this kernel, we are also given the pressure values. Thus, we also create a local copy of the pressure matrix and put the pressure values into registers.

```
1   __local myfloat Lp[Nq][Nq+PAD];
2
3   myfloat pk[Nq]; //register
4
5   for(k=0;k<Nq;++k) {
6     pk[k] = pL[id];
7     id += Nq*Nq;
8   }
9
10  for(k=0;k<Nq;++k){
11    barrier(CLK_LOCAL_MEM_FENCE);
12
13    // share pressure
14    Lp[j][i] = pk[k];
```

Then, because $\tilde{u}$, $\tilde{v}$ and $\tilde{w}$ are given inputs, the registers are populated with this information.

```
1     id = e*BSIZE+k*Nq*Nq+j*Nq+i;
2     const myfloat uk = utildeL[id];
3     const myfloat vk = vtildeL[id];
4     const myfloat wk = wtildeL[id];
```

Next, using the differentiation matrix, we find the partial derivative of the pressure, $\frac{\partial p}{\partial(r,s,t)}$, which is analagous to Equation (3.1):

$$\left(\frac{\partial p}{\partial r}\right)_{ijke} = D_{im}p^e_{mjk}, \quad \left(\frac{\partial p}{\partial s}\right)_{ijke} = D_{jm}p^e_{imk}, \quad \left(\frac{\partial p}{\partial t}\right)_{ijke} = D_{km}p^e_{ijm}.$$

```
1      myfloat pr = 0.f, ps = 0.f, pt = 0.f;
2
3      for(int m=0;m<Nq;++m) pt += LD[k][m]*pk[m];
4      for(int m=0;m<Nq;++m) pr += LD[i][m]*Lp[j][m];
```

```
5       for(int m=0;m<Nq;++m) ps += LD[j][m]*Lp[m][i];
```

Finally, we find the gradient of the pressure, which we defined in Equation (2.33):

$$\nabla p^{n+1} = \begin{pmatrix} \frac{\partial p}{\partial x} & \frac{\partial p}{\partial y} & \frac{\partial p}{\partial z} \end{pmatrix}^T,$$

where

$$\begin{aligned} \frac{\partial p}{\partial x} &= \frac{\partial r}{\partial x}\frac{\partial p}{\partial r} + \frac{\partial s}{\partial x}\frac{\partial p}{\partial s} + \frac{\partial t}{\partial x}\frac{\partial p}{\partial t} \\ \frac{\partial p}{\partial y} &= \frac{\partial r}{\partial y}\frac{\partial p}{\partial r} + \frac{\partial s}{\partial y}\frac{\partial p}{\partial s} + \frac{\partial t}{\partial y}\frac{\partial p}{\partial t} \\ \frac{\partial p}{\partial z} &= \frac{\partial r}{\partial z}\frac{\partial p}{\partial r} + \frac{\partial s}{\partial z}\frac{\partial p}{\partial s} + \frac{\partial t}{\partial z}\frac{\partial p}{\partial t}. \end{aligned}$$

Then, we subtract ∇p^{n+1} from $\frac{\tilde{u}}{dt}$, and we incorporate the weights and the Reynolds number.

$$\begin{aligned} \tilde{\tilde{u}} &\approx w_i w_j w_k J^e_{ijk} \left(\tilde{u} - dt\, \nabla p^{n+1}\right) \\ &= \widehat{\mathbf{W}}(i,j,k) \left(\tilde{u} - dt\, \nabla p^{n+1}\right) \end{aligned}$$

Again, `wJ` is the entry in $\widehat{\mathbf{W}}$ associated with each GLL node.

```
1       id = e*BSIZE+k*Nq*Nq+j*Nq+i;
2
3       utilde2L[id] = Re*wJ*(dtinv*uk - (rx*pr+sx*ps+tx*pt));
4       vtilde2L[id] = Re*wJ*(dtinv*vk - (ry*pr+sy*ps+ty*pt));
5       wtilde2L[id] = Re*wJ*(dtinv*wk - (rz*pr+sz*ps+tz*pt));
```

Thus, the final value for $\tilde{\tilde{u}}$ is the inner product

$$\tilde{\tilde{u}} = \left(\phi, \tilde{u} - dt\, \nabla p^{n+1}\right)_\Omega.$$

3.4 Diffusion Step

In the diffusion step, we must solve the following screened Coulomb potential problem,

$$\mathbf{A}\, u^{n+1} = -\frac{1}{\nu dt}\widehat{\mathbf{W}}\, \tilde{\tilde{u}},$$

where $\mathbf{A} = \mathbf{L} + \frac{1}{\nu dt}\mathbf{W}$. To do so, we execute the preconditioned conjugate gradient method described in the following section. After finding the solution to the above equation, we ensure that the nodes on the Dirichlet boundaries have the proper value using the `Boundary.cl` kernel.

3.4.1 Boundary Kernel

The `Boundary.cl` kernel takes `bcType`, `Fmask`, u, v, w, p, x, y, z, and t as kernel arugments. Here `bcType` has nonzeros entries that correspond to the boundary nodes. `Fmask` contains the local node numbers of nodes on faces of an element. Then, x, y, and z are vectors that contain the coordinates of each node, and t is the current time.

```
1  __kernel void Boundary(__global int* bcType, __global int*
       Fmask,
2                         __global myfloat* u,  __global
                             myfloat* v,
3                         __global myfloat* w,  __global
                             myfloat* p,
4                         __global const myfloat* x,
5                         __global const myfloat* y,
6                         __global const myfloat* z,
7                         const myfloat t){
```

We first assign each element to a work group, and then assign IDs to each work item in a work group. In addition, we initialize the values for the global x, y, and z coordinates, the boundary condition identifier, and the face and global IDs.

```
1    int e = get_group_id(0);
2    int i = get_local_id(0);
3    int j = get_local_id(1);
4
5    myfloat gx, gy, gz;
6    int bc, fid, gid;
```

Then, we loop through each of the faces on an element, and set the boundary condition identifier. If the current node is on the boundary, we set the local node ID and then find the global ID using `Fmask`. From here, we can extract the global coordinates of the boundary node.

```
1     for(int f=0; f<6; f++){
2       bc = bcType[6*e + f];
3
4       if(bc){
5         fid = i + j*Nq + f*Nq2;
6         gid = e*Nq3 + (Fmask[fid] - 1);
7
8         gx = x[gid];
9         gy = y[gid];
10        gz = z[gid];
11
12        // Edges/Corners will eventually overlap
13        //   so the barrier will prevent overwrites
14        barrier(CLK_GLOBAL_MEM_FENCE);
```

Once we have the coordinates for the boundary node, we set the corresponding entry in u, v or w to be the Dirichlet boundary condition defined in `u_bc`, `v_bc`, or `w_bc`, respectively. Finally, we check to see if the boundary node lies on a Dirichlet pressure boundary. If so, we set the Dirichlet value in the corresponding entry of p.

```
1         if(bc & vDirichletBC){
2           u[gid] = u_bc(bc, gx, gy, gz, t);
3           v[gid] = v_bc(bc, gx, gy, gz, t);
4           w[gid] = w_bc(bc, gx, gy, gz, t);
5         }
6         if(bc & pDirichletBC)
7           p[gid] = p_bc(bc, gx, gy, gz, t);
8       }
9     }
```

3.5 Screened Coulomb Problem

In this section, we describe the implementation of the method used to solve the screened Coulomb problem, which requires us to solve the following PDE:

$$(-\Delta + \lambda)u = f$$

$$\text{or} \quad Au = f.$$

To solve this system, we implement a preconditioned conjugate gradient method, which we introduced in Section 2.3.1. Thus, we first provide the algorithm for the preprocessing step, where the commented part of the following pseudocode indicates whether the computation is done using a specific kernel or on the CPU.

Algorithm/Pseudocode for Preprocessing Step

```
1  // PCGpart1_local.cl
2  Apply stiffness matrix to initial guess scp_uL
3  Add result to scp_FL
4  scp_FL += A*scp_uL
5
6  // Gather.cl
7  Gather scp_FL into scp_r
8
9  // PCGpart5a.cl
10 Apply preconditioner to scp_r
11 scp_zP = P*scp_r
12
13 // Gather.cl
14 Gather scp_zP into scp_z
15 Gather scp_zP into scp_p
16
17 // PCGpart3.cl
18 Compute partial reduction of scp_r*scp_z
19
20 // On the CPU
21 Finish reduction of rdotzold = scp_r*scp_z
```

```
22
23 r2 = rdotzold
```

Next, we provide the algorithm for each iteration of the conjugate gradient method. Again, the commented part of the pseudocode indicates whether the computation is done using a specific kernel or on the CPU.

Algorithm/Pseudocode for Preconditioned Conjugate Gradient Iterations

```
1  while(true){
2
3    // PCGpart1.cl
4    Apply stiffness matrix to scp_p
5    scp_ApL = A*scp_p
6
7    // PCGpart2.cl
8    Gather scp_ApL into scp_Ap
9    Compute partial reduction of scp_p*scp_Ap
10
11   // On the CPU
12   Finish reduction of pAp = scp_p*scp_Ap
13
14   // If all Neumann Boundary conditions
15   Execute constant correction
16
17   // On the CPU
18   alpha = rdotzold/pAp
19
20   // PCGpart4.cl
21   Compute scp_u = scp_u + alpha*scp_p
22   Compute scp_r = scp_r - alpha*scp_Ap
23   Compute partial reduction of scp_r*scp_r
24
25   // On the CPU
26   Finish reduction of r2 = scp_r*scp_r
27   Check for convergence
28   if(r2 < tol) break;
29
30   // PCGpart5a.cl
31   Apply preconditioner to scp_r
```

```
32    scp_zP = P*scp_r
33
34    // PCGpart5b.cl
35    Gather scp_zP into scp_z
36    Compute partial reduction of scp_r*scp_z
37
38    // On the CPU
39    Finish reduction of rdotznew = scp_r*scp_z
40    beta = rdotznew/rdotzold;
41    rdotzold = rdotznew;
42
43    // PCGpart5c.cl
44    Compute scp_p = scp_z + beta*scp_p
45
46    // Scatter.cl
47    Scatter scp_u to scp_ApL
48
49    // Add.cl
50    Compute scp_uL = scp_uL + scp_ApL
51 }
```

In the following sections, we explain each of the kernels listed in the above algorithms/pseudocodes, and refer the reader to Section 3.2.2 for explanation of the Gather/Scatter kernels.

3.5.1 PCGPart1_local

The `PCGpart1_local.cl` kernel applies the stiffness matrix A to the initial guess u. This kernel executes the same computations as the `PCGpart1.cl` kernel, except that the stiffness matrix is applied to each local node of u, rather than the global nodes. That is, `galnums` is not a kernel argument, and thus we copy the entire vector p into register and local memory. Thus, we only show the code in `PCGpart1_local.cl` that differs from `PCGpart1.cl` below, and refer the reader to Section 3.5.2 for the details

in computing Ap. First, the kernel arguments simply do not include `galnums`, and the resulting vector is instead called `FL`.

```
1  __kernel void PCGpart1_local(const int K, const myfloat
       lambda,
2                               __global const myfloat *
                                   restrict geo,
3                               __constant myfloat * restrict D,
4                               __global const myfloat *
                                   restrict u,
5                               __global myfloat * restrict FL){
```

Then, we load the entire u vector into registers, which will later also be loaded into local memory.

```
1    /* *** Change load here to use scatter array *** */
2    /* load pencil of u into register */
3    unsigned int id = e*BSIZE+j*Nq+i;
4
5  #if UNROLL==1
6  #pragma unroll 16
7  #endif
8    for(k=0;k<Nq;++k) {
9      id = e*BSIZE+k*Nq*Nq+j*Nq+i;
10
11     uk[k] = u[id];
12
13     lapu[k] = 0.f;
14   }
15
16   barrier(CLK_LOCAL_MEM_FENCE);
```

3.5.2 PCGPart1

The `PCGpart1.cl` kernel takes K, λ, `galnums`, `geo`, `D`, u, and `NL` as kernel arguments, and applies the stiffness matrix to the current guess for the solution, u.

```
1  __kernel void PCGpart1(const int K, const myfloat lambda,
2                         __global const int * restrict galnums,
```

```
3                           __global const myfloat * restrict geo,
4                           __constant myfloat * restrict D,
5                           __global const myfloat * restrict u,
6                           __global myfloat * restrict NL){
```

From Equation (2.15), we see that the stiffness matrix can be defined as $A = D^T G D$, where

$$D = \begin{pmatrix} D_r \\ D_s \\ D_t \end{pmatrix} \quad \text{and} \quad G = \begin{pmatrix} G_{rr} & G_{rs} & G_{rt} \\ G_{sr} & G_{ss} & G_{st} \\ G_{tr} & G_{ts} & G_{tt} \end{pmatrix}.$$

The matrix `geo` contains the values for each entry in G. Thus, computing Au begins with the tensor product derivative evaluations of u in each (r, s, t) direction [9]. Then we apply the geometric factors in G to the components of u, and finally, apply the transposed derivative matrix to u [9]. In sum, applying the stiffness matrix A to the vector u becomes

$$A\,u = \sum_i D_i^T \left(\sum_j G_{ij} D_j u \right).$$

To do this, we first allocate local memory as described in Section 3.1 (and thus omit the code) for the differentiation matrix and the components u, assigning each element to a work group and node to a work item. However, when allocating registers for u, we also allocate registers for the vector Au, `lapu`.

```
1     // u[:][j][i] -> uk[:]
2     myfloat   uk[Nq];
3     myfloat lapu[Nq]; // use shared ?
```

Then, in order to use global version of u, we use `galnums` to load the global values of u into the registers.

```
    /* *** Change load here to use scatter array *** */
    /* load pencil of u into register */
    unsigned int id = e*BSIZE+j*Nq+i;

  #if UNROLL==1
  #pragma unroll 16
  #endif
    for(k=0;k<Nq;++k) {
      id = e*BSIZE+k*Nq*Nq+j*Nq+i;
      int gid = galnums[id];

      myfloat tmp = 0.0;

      if(gid >= 0)
        tmp = u[gid]; // this becomes uncoalesced but ** maybe
            ** cached

      uk[k] = tmp;
      lapu[k] = 0.f;
    }

    barrier(CLK_LOCAL_MEM_FENCE);
```

Finally, we also prefetch the geometric factors in G in a similar manner to that of Section 3.1. That is, we assign values to `G00`, `G01`, `G02`, `G11`, `G12`, `G22`, and `J`, where `J` is the value of the Jacobian at each node. These values correspond to G_{rr}, G_{rs}, G_{rt}, G_{ss}, G_{st}, and G_{tt}, respectively. In the same for loop, we also initialize the derivatives $\frac{\partial u}{\partial(r,s,t)}$.

```
#if UNROLLOUTER==1
#pragma unroll 16
#endif
  for(k=0;k<Nq;++k){
    // prefetch geometric factors
    id = 7*e*BSIZE+k*Nq*Nq+j*Nq+i;

```

```
8       const myfloat G00 = geo[id]; id+= BSIZE;
9       const myfloat G01 = geo[id]; id+= BSIZE;
10      const myfloat G02 = geo[id]; id+= BSIZE;
11      const myfloat G11 = geo[id]; id+= BSIZE;
12      const myfloat G12 = geo[id]; id+= BSIZE;
13      const myfloat G22 = geo[id]; id+= BSIZE;
14      const myfloat J   = geo[id];
15
16      Lu[j][i] = uk[k];
17
18      myfloat ur = 0.f;
19      myfloat us = 0.f;
20      myfloat ut = 0.f;
```

As in previous kernels, to first compute the derivatives $\frac{\partial u}{\partial(r,s,t)}$ we use the differentiation matrix:

$$\left(\frac{\partial u}{\partial r}\right)_{ijke} = D_{im}u^e_{mjk}, \quad \left(\frac{\partial u}{\partial s}\right)_{ijke} = D_{jm}u^e_{imk}, \quad \left(\frac{\partial u}{\partial t}\right)_{ijke} = D_{km}u^e_{ijm}.$$

```
1   #if UNROLL==1
2   #pragma unroll 16
3   #endif
4       for(int m=0;m<Nq;++m) ut += LD[k][m]*uk[m];
5
6       barrier(CLK_LOCAL_MEM_FENCE);
7
8   #if UNROLL==1
9   #pragma unroll 16
10  #endif
11      for(int m=0;m<Nq;++m) ur += LD[i][m]*Lu[j][m];
12
13  #if UNROLL==1
14  #pragma unroll 16
15  #endif
16      for(int m=0;m<Nq;++m) us += LD[j][m]*Lu[m][i];
```

Then, if we define $\bar{u} = (u_r\, u_s\, u_t)$, applying the geometric factors, we see that

$$\hat{u} = G\,\bar{u} = \sum_j G_{ij}\bar{u}.$$

This value is written into the corresponding entries of `Lw` and `Lv`. Note that the operator A in Equation (3.5) includes the scalar λ. Thus, we also add λu into the summation.

```
1      Lw[j][i] = G01*ur + G11*us + G12*ut;
2      Lv[j][i] = G00*ur + G01*us + G02*ut;
3
4      // put this here for a performance bump
5      const myfloat GDut = G02*ur + G12*us + G22*ut;
6
7      myfloat lapuk = J*(lambda*uk[k]);
8
9      barrier(CLK_LOCAL_MEM_FENCE);
```

The last step is to sum across the transposed derivatives. That is, we apply D^T to $\hat{u}$:

$$A\,u = \sum_i D_i^T \bar{u}.$$

Again, we use the differentiation matrix, and in order to incorporate the transpose, we simply switch the indices:

$$\left(\frac{\partial u}{\partial r}\right)_{ijke} = D_{mi} u^e_{mjk}, \quad \left(\frac{\partial u}{\partial s}\right)_{ijke} = D_{mj} u^e_{imk}, \quad \left(\frac{\partial u}{\partial t}\right)_{ijke} = D_{mk} u^e_{ijm}.$$

```
1  #if UNROLL==1
2  #pragma unroll 16
3  #endif
4      for(int m=0;m<Nq;++m)
5        lapuk += LD[m][j]*Lw[m][i];
6
7  #if UNROLL==1
8  #pragma unroll 16
9  #endif
10     for(int m=0;m<Nq;++m)
11       lapu[m] += LD[k][m]*GDut; // DT(m,k)*ut(i,j,k,e)
12
13 #if UNROLL==1
14 #pragma unroll 16
```

```
#endif
    for(int m=0;m<Nq;++m)
      lapuk+= LD[m][i]*Lv[j][m];

    lapu[k] += lapuk;
  }
```

Finally, adding each of the transposed derivatives to the value of `lapu`, we populate `NL` with the summation in Equation (3.5.2).

```
#if UNROLL==1
#pragma unroll 16
#endif
  for(k=0;k<Nq;++k){
    id = e*BSIZE+k*Nq*Nq+j*Nq+i;
    NL[id] = lapu[k];
  }

}
```

3.5.3 PCGPart2

The `PCGpart2.cl` kernel takes N, `starts`, `indices`, `ApL`, `Ap`, p, and `red` as arguments.

The `PCGpart2.cl` kernel then computes a partial reduction of $p \cdot Ap$.

```
__kernel void PCGpart2(const int N,
                       __global const int * restrict starts,
                       __global const int * restrict indices,
                       __global const myfloat * restrict ApL,
                       __global myfloat * restrict Ap,
                       __global myfloat * restrict p,
                       __global myfloat * restrict red
                       ){
```

We first allocate local memory for the vector to reduce, `s_a`. Then, using the local ID we define the local work item ID to be `tx`, and using the global ID we define the global work item ID to be `n`. Thus, `tx` is the ID of the work item for a single

work group. That is, `tx` is independent of the number of work groups used in the computation.

```
1   // find coordinate of thread in global axis-0
2   volatile __local myfloat s_a[bdim];
3   int tx = get_local_id(0);
4   s_a[tx] = 0;
5
6   int n = get_global_id(0);
```

As in the `Gather.cl` kernel, we use the `starts` and `indices` vectors to sum the local entries in `ApL` that correspond to each global node.

```
1   if(n<N){
2     int start = starts[n];      // coalesced
3     int end   = starts[n+1];    // coalesced ?
4
5     myfloat gun = 0;
6     myfloat pn = p[n];
7
8     for(int m=start;m<end;++m){
9       const int ind = indices[m]; // not coalesced
10      gun += (ind>=0) ? ApL[ind] : 0;
11    }
```

This summation then becomes the n^{th} entry in the global vector `Ap`. From here, we simply multiply the n^{th} entry of `Ap` with the n^{th} entry of p, and then use the `workgroup_reduce.hpp` file to sum all the entries in `s_a`.

```
1     // is thread in range for the addition
2     Ap[n] = gun;
3
4     // assume bx power of 2
5     s_a[tx] = gun*pn;
6   }
7   // reduce s_a to one entry by addition
8   workgroup_reduce(tx, s_a);
```

Finally, we set the entries of `red` to be the summation found above.

```
  // final reduction
  int bx = get_group_id(0);
  if(tx==0)
    red[bx] = s_a[0];
}
```

3.5.4 PCGPart3

The `PCGpart3.cl` kernel takes any two vectors and computes the partial reduction for the dot product of those two vectors. That is, `PCGpart3.cl` takes N, a, b, and `rab` as kernel arguments, where `rab` is the reduced dot product of a and b.

```
__kernel void PCGpart3(int N,
          __global const myfloat * restrict a,
          __global const myfloat * restrict b,
          __global myfloat * restrict rab){
```

We again allocate `s_a` in local memory, and assign work item, work group, and local work item IDs, along with the global size.

```
  volatile __local myfloat s_a[bdim];

  // assume a 1 dimensional thread array
  int n = get_global_id(0);
  int tx = get_local_id(0);
  int bx = get_group_id(0);
  int gx = get_global_size(0);
```

Then, we simply loop through and sum the products of each entry in a with the corresponding entry in b. This summation then becomes the appropriate entry in `s_a`, which is sent to `workgoup_reduce.hpp`.

```
  // is thread in range for the addition
  myfloat d = 0.f;
  while(n<N){
```

```
4       d += a[n]*b[n];
5       n += gx;
6     }
7     // assume bx power of 2
8     s_a[tx] = d;
9
10    // reduce s_a to one entry by addition
11    workgroup_reduce(tx, s_a);
```

Again, the reduction is finished by placing the summation found in `workgoup_reduce.hpp` in each of the entries in rab.

```
1     // final reduction
2     if(tx==0)
3       rab[bx] = s_a[0];
```

3.5.5 PCGPart4

The `PCGpart4.cl` kernel performs three parts of a conjugate gradient iteration. That is, this kernel computes

$$u = u + \alpha p \quad \text{and} \quad r = r - \alpha Ap,$$

and then performs the partial reduction of $r \cdot r$. The kernel arguments for `PCGpart4.cl` are `Unique`, or the degrees of freedom, α, Ap, p, u, r, and `red`.

```
1   __kernel void PCGpart4(const int Unique,
2                          const myfloat alpha,
3                          __global const myfloat * restrict Ap,
4                          __global const myfloat * restrict p,
5                          __global myfloat * restrict u,
6                          __global myfloat * restrict r,
7                          __global myfloat * restrict red){
```

We begin, as usual, by allocating local memory for the reduced vector `s_a`, and assigning IDs to each of the local work items.

```
1   volatile __local myfloat s_a[bdim];
2   int tx = get_local_id(0);
3   s_a[tx] = 0;
4
5   // assume a 1 dimensional thread array
6   int n = get_global_id(0);
```

We also set `n` to be the global ID of each work item, and use this to compute $r - \alpha Ap$ at each entry in r. Then, we populate `s_a` with the squares of each entry in r.

```
1   if(n<Unique){
2     u[n] += alpha*p[n];
3     const myfloat rn = r[n] - alpha*Ap[n];
4
5     r[n] = rn;
6
7     // assume bx power of 2
8     s_a[tx] = rn*rn;
9   }
```

Finally, we perform the partial reduction of $r \cdot r$. That is, after performing a work group reduction on `s_a` we put each of the summations into the corresponding entry in `red`.

```
1   // reduce s_a to one entry by addition
2   workgroup_reduce(tx, s_a);
3
4   // final reduction
5   int bx = get_group_id(0);
6   if(tx==0)
7     red[bx] = s_a[0];
```

3.5.6 PCGPart5a

The `PCGpart5a.cl` kernel takes `elements`, λ, `scp_galnumsP`, `scp_h`, `scp_PVL`, `scp_PVR`, `scp_PWR`, `scp_wp`, `scp_r`, and `scp_zP` as parameters. Here `scp_galnumsP` is the ma-

trix that maps local node numbers to global numbers as discussed in Section 2.1.3. However, `scp_galnumsP` maps the preconditioner nodes, for which there is an overlap between elements, as discussed in Section 2.3.1 (see Figure 2.4). Then, `scp_h` is a vector that corresponds to the element size, which has the x, y, and z lengths of the element for entries.

```
1  __kernel void PCGpart5a(const int K, const myfloat lambda,
2                  __global const int      * restrict galnums,
3                  __global const myfloat4 * restrict h,
4                  __global const myfloat  * restrict PVL,
5                  __global const myfloat  * restrict PVR,
6                  __global const myfloat  * restrict PWR,
7                  __global const myfloat  * restrict  w,
8                  __global const myfloat  * restrict u,
9                  __global myfloat *  restrict Pu){
```

Next, in order to compute the inverse preconditioner matrix, as discussed in Section 2.3.1, `scp_PVL`, `scp_PVR`, and `scp_PWR` are the matrix whose columns are the left eigenvectors, the matrix whose columns are the right eigenvectors, and the vector of eigenvalues, respectively. Then, `scp_wp` are the weights; `scp_r` and `scp_zP` are the current r and zP values.

We first allocate local memory for the eigenvectors, eigenvalues, and weights. Note that the size of these matrices is $NqP \times NqP \times K$, where NqP is the number of nodes for the local subdomains, $\bar{D}^e$, used in the preconditioner. That is, $NqP = Nq + 2$ which includes one layer of GLL nodes outside of each element D^k.

```
1     __local myfloat LPVR[NqP][NqP+PADP];
2     __local myfloat LPVL[NqP][NqP+PADP];
3     __local myfloat LPWR[NqP];
4
5     __local myfloat LPu[NqP][NqP+PADP]; // to reuse or not
```

```
6
7     __local myfloat Lw[NqP];
8
9     // u[:][j][i] -> uk[:]
10    myfloat  Puk[NqP]; // use shared ?
11    myfloat bcuk[NqP];
12  #if LOCALSTORE==1
13    __local myfloat LPPuk[NqP][NqP][NqP]; // use shared ?
14  #else
15    myfloat PPuk[NqP];
16  #endif
```

Then, as discussed in Section 3.1, we assign each element to a work group using the group ID, and then assign each node to a work item using the local IDs.

```
1     const unsigned int e = get_group_id(0);
2     const unsigned int i = get_local_id(0);
3     const unsigned int j = get_local_id(1);
4     unsigned int k;
```

After assigning the work groups and work items, we then load the respective values for the right eigenvectors, left eigenvectors, eigenvalues, and weights into `LPVR`, `LPVL`, `LPWR`, `Lw`, respectively.

```
1     /* load into local memory */
2     LPVR[j][i] = PVR[NqP*i+j]; // PVR is column major
3     LPVL[j][i] = PVL[NqP*i+j]; // PVL is column major
4
5     if(j==0) LPWR[i] = PWR[i];
6     if(j==0) Lw[i] = w[i];
7
8     unsigned int id = e*BSIZEP+j*NqP+i;
9     int m;
```

After allocating and populating the required eigenvectors, eigenvalues and weights, we initialize the preconditioned values `Puk` and `PPuk` to zero.

```
1     for(m=0;m<NqP;++m) {
2       Puk[m] = 0;
```

```
3   #if LOCALSTORE==1
4       LPPuk[m][j][i] = 0;
5   #else
6       PPuk[m] = 0;
7   #endif
8     }
```

Then, similar to `PCGpart1.cl`, we use `galnums` to load the global values of u onto the registers. Note that we also create `he`, which gives the element's length in each (r, s, t) direction, and the Jacobian value. In this section, we also incorporate the tensor product weights into the velocity, $u = w_i w_j w_k u$.

```
1     // mesh sizes (broadcast?)
2     const myfloat4 he = h[e];
3
4     barrier(CLK_LOCAL_MEM_FENCE);
5
6     for(k=0;k<NqP;++k) {
7       int gid = galnums[id];
8       myfloat uk;
9
10      if(gid >= 0){
11        uk      = u[gid]; // not coalesced
12        bcuk[k] = 1;
13      }
14
15      uk = uk*(Lw[i]*Lw[j]*Lw[k]);
```

Now that the weights have been incorporated, we begin the transformations using the eigenvectors and eigenvalues. That is, we first transform in t, and then transfrom in r and s. These transformations are done by multiplying the weighted u by the left eigenvectors. Once we have transformed in all three directions, we will have transformed `Puk` on a single (r, s) plane.

```
1       // transform in t
2       for(m=0;m<NqP;++m) Puk[m] += LPVL[m][k]*uk;
```

```
3
4       id += NqP*NqP;
5     }
6
7     // transform in r then s
8     for(k=0;k<NqP;++k){
9
10      barrier(CLK_LOCAL_MEM_FENCE);
11      LPu[j][i] = Puk[k];
12      barrier(CLK_LOCAL_MEM_FENCE);
13
14      {
15        myfloat tmp = 0.f;
16
17        // transform in r
18        for(m=0;m<NqP;++m) tmp += LPVL[i][m]*LPu[j][m];
19
20        barrier(CLK_LOCAL_MEM_FENCE);
21        LPu[j][i] = tmp;
22      }
23
24      barrier(CLK_LOCAL_MEM_FENCE);
25
26      {
27        myfloat tmp = 0.f;
28
29        // transform in s
30        for(m=0;m<NqP;++m) tmp += LPVL[j][m]*LPu[m][i];
```

Thus, after performing these transformations, we have

$$(\mathbf{P}^e)^{-1} = \tilde{S}_r \otimes \tilde{S}_s \otimes \tilde{S}_t \left(I \otimes I \otimes \tilde{\Lambda}_r + I \otimes \tilde{\Lambda}_s \otimes I + \tilde{\Lambda}_t \otimes I \otimes I \right),$$

where we have neglected the geometry of the element. Thus, in the next step, we incorporate the lengths of the element in each direction into the right eigenvectors. Then, we ensure that the value of `LPu` in shared memory incorporates the geometry of the element.

```
1       // at this point Puk is totally transformed on level k
```

```
2         tmp /= he.w*(LPWR[k]*(he.z*he.z) + LPWR[j]*(he.y*he.y)
              + LPWR[i]*(he.x*he.x) + lambda);
3
4         barrier(CLK_LOCAL_MEM_FENCE);
5
6         // load into shared again
7         LPu[j][i] = tmp;
8       }
9
10      barrier(CLK_LOCAL_MEM_FENCE);
11
12      {
13        myfloat tmp = 0.f;
```

Then, to get the correct value for the preconditioner matrix, we must tranform back in each direction, or multiply by the right eigenvectors:

$$\mathbf{A}^{-1} = S_r \otimes S_s \otimes S_t \left(I \otimes I \otimes \Lambda_r + I \otimes \Lambda_s \otimes I + \Lambda_t \otimes I \otimes I\right) S_r^{-1} \otimes S_s^{-1} \otimes S_t^{-1}.$$

```
1         // transform back in r
2         for(m=0;m<NqP;++m) tmp += LPVR[i][m]*LPu[j][m];
3
4         barrier(CLK_LOCAL_MEM_FENCE);
5
6         // load into shared again
7         LPu[j][i] = tmp;
8       }
9       barrier(CLK_LOCAL_MEM_FENCE);
10      {
11        myfloat tmp = 0.f;
12
13        // transform back in s
14        for(m=0;m<NqP;++m) tmp += LPVR[j][m]*LPu[m][i];
15
16        // transform back in t
17  #if LOCALSTORE==1
18        for(m=0;m<NqP;++m) LPPuk[m][j][i] += LPVR[m][k]*tmp;
19  #else
20        for(m=0;m<NqP;++m) PPuk[m] += LPVR[m][k]*tmp;
21  #endif
22      }
```

```
23     }
```

Lastly, we write the final transformed value to the corresponding entries in `Pu`, which is the the preconditioned version of u.

```
1     for(k=0;k<NqP;++k){
2       id = e*BSIZEP+k*NqP*NqP+j*NqP+i;
3
4   #if LOCALSTORE==1
5       Pu[id] = LPPuk[k][j][i]*bcuk[k];
6   #else
7       Pu[id] = PPuk[k]*bcuk[k];
8   #endif
```

3.5.7 PCGPart5b

The `PCGpart5b.cl` kernel gathers `Zp` to z and begins the reduction of $r \cdot z$. The kernel arguments are N, `starts`, `indices`, `ApL`, `Ap`, p, and `red`.

```
1  __kernel void PCGpart5b(const int N,
2                          __global int * restrict starts,
3                          __global int * restrict indices,
4                          __global myfloat * restrict ApL,
5                          __global myfloat * restrict Ap,
6                          __global myfloat * restrict p,
7                          __global myfloat * restrict red
8                          ){
```

We then go through the same routine of allocating local memory for `s_a` and setting the local and global work item IDs.

```
1    // find coordinate of thread in global axis-0
2    volatile __local myfloat s_a[bdim];
3    int tx = get_local_id(0);
4    s_a[tx] = 0;
5
6    int n = get_global_id(0);
```

From there, we perform the gather operation described in Section 3.2.2 on `ApL`, and the gathered entries are written to the vector `Ap`.

```
  if(n<N){
    int start = starts[n];      // coalesced
    int end   = starts[n+1];    // coalesced ?

    myfloat gun = 0;
    myfloat pn = p[n];

    for(int m=start;m<end;++m){
      const int ind = indices[m]; // not coalesced
      gun += (ind>=0) ? ApL[ind] : 0;
    }

    // is thread in range for the addition
    Ap[n] = gun;
```

Then, to begin the dot product $r \cdot z$, we populate `s_a` with the products of the entries in `Ap` and p.

```
    // assume bx power of 2
    s_a[tx] = gun*pn;
  }
```

Finally, we perform the workgroup reduction and finish the reduction in the `PCGpart5b.cl` kernel.

```
  // reduce s_a to one entry by addition
  workgroup_reduce(tx, s_a);

  // final reduction
  int bx = get_group_id(0);
  if(tx==0)
    red[bx] = s_a[0];
```

3.5.8 PCGPart5c

The last part of a conjugate gradient iteration is performed by the `PCGpart5c.cl` kernel. Specifically, this kernel computes

$$p = z + \beta p.$$

Thus, we send `Unique` (degrees of freedom), z, β, and p as kernel arguments.

```
__kernel void PCGpart5c(const int Unique,
          global const myfloat * restrict z,
          const myfloat beta,
          global myfloat * restrict p){
```

From this information, the kernel simply assigns each work item to a degree of freedom, and performs the computation above for each entry in the vector p.

```
  // assume a 1 dimensional thread array
  int n = get_global_id(0);

  if(n>=Unique) return;

  p[n] = z[n] + beta*p[n];

}
```

3.5.9 Pressure Constant Correction in Preconditioned CG

We use the preconditioned conjugate gradient method to solve a linear equation of the form $Ax = b$. The solution of this equation may not be unique, even if b is in the range of A. The uniqueness in the solution is lost when A is a singular matrix. Thus, to avoid singular matrices, we shift the spectrum of A away from zero. That is, as discussed in section 2.3, we set $\hat{A}x = \left(A + \vec{1} \cdot \vec{1}^T\right) x$. Then, the system we must solve

is $\hat{A}x = b$, which is nonsingular, and has a unique solution. Before every iteration of the conjugate gradient method, we first check if we are solving a screened Coulomb potential problem with all Neumann boundary conditions.

```
1   // Pressure Correction
2   if(constantCorrection){
```

Then, we reduce the local u matrix, `scp_uL`, which is the current x. This requires first a kernel call to `PressureReduce.cl`, which returns a vector. We then finish the reduction on the CPU, to get the final value of $sum(x) = \vec{1} \cdot \vec{1}^T x$. In the code, `Pred` $= sum(x)$.

```
1   // First, partial reduce of u
2   PressureReduce(unique, scp_uL, scp_red);
3
4   // Complete reduce of u
5   helper.tohost(Nred*sizeof(myfloat),red.c_array(),scp_red);
6   myfloat Pred = 0.0;
7   for(n=1; n<=Nred; n++)
8    Pred += red(n);
```

Finally, we call the kernel `ScalarAdd.cl`, which adds a scalar value to a vector. Here, we are completing $Ax + sum(x)$, where in the code, $Ax =$ `scp_Ap`.

```
1   ScalarAdd(unique, scp_Ap, Pred);
```

Thus, after the kernel call to `ScalarAdd`, we have completed the constant correction and now have $\hat{A}x = Ax + \vec{1} \cdot \vec{1}^T x$, which translates to `scp_Ap = scp_Ap + Pred`.

In the original constant correction, we subtracted the constant out of the solution vector, x, in the preconditioner and at each iteration of the conjugate gradient method. Similar to previous method, we first reduce a vector. In this version of the constant correction, we must first reduce `scp_FL` in the preconditioning step. Thus,

we call the kernel `PressureReduce.cl`, and then again finish the reduction on the CPU, where $sum(x)$ =`Pred`.

```
1   // Pressure Correction
2   if(constantCorrection){
3   // First, partial reduce of u
4    PressureReduce(total, scp_FL, scp_red);
5
6    // Complete reduce of u
7    helper.tohost(Nred*sizeof(myfloat), red.c_array(), scp_red
        );
8    myfloat Pred = 0.0;
9    for(n=1; n<=Nred; n++)
10    Pred += red(n);
```

Next, in order to compute $\frac{sum(x)}{n}$, we must divide `Pred` by the degrees of freedom, which in the code is `unique`. Finally, we subtract this value from `scp_FL` to get the new solution vector, which is the initial guess in the preconditionr: $x = \hat{x} - \frac{sum(x)}{n}$.

```
1   Pred /= ((myfloat) unique);
2   ScalarAdd(unique, scp_FL, Pred);
```

Next, after each iteration of the conjugate gradient method, we must perform the same steps. In this case, however, we are reducing `scp_Ap`, and subtracting the constant from `scp_u`.

```
1   // Pressure Correction
2   if(constantCorrection){
3    // First, partial reduce of u
4    PressureReduce(unique, scp_Ap, scp_red);
5
6    // Complete reduce of u
7    helper.tohost(Nred*sizeof(myfloat),red.c_array(),scp_red);
8    myfloat Pred = 0.0;
9    for(n=1; n<=Nred; n++)
10    Pred += red(n);
11   Pred /= ((myfloat) unique);
12   ScalarAdd(unique, scp_u, Pred);}
```

Thus, we have $x = \hat{x} - \frac{sum(x)}{n}$, which translates to `scp_u` = `scp_u` - sum(`scp_Ap`)/`unique` in the code.

Chapter 4

Test Cases

In verifying the solver, we use several different test cases. We first use a one dimensional steady state solution to the INS equations known as Channel flow. We use another one dimensional steady state solution where the the boundary conditions dictate the velocity, known as Shear flow. Next, we use a two dimensional steady state solution to the INS equations, known as Kovasznay flow. Finally, we implement an analytical lid driven cavity flow test case.

4.1 Channel Flow

In this test case we examine Channel flow, which describes a 2D steady viscous fluid flow between two plates under a constant pressure gradient. For this particular test, we define the domain, Ω, by $x = [-1, 1]$ and $y = z = [-0.5, 0.5]$. We can find the exact solution to the INS equations by examining the properties of this flow. We first note that because this is a steady state flow,

$$\frac{\partial u}{\partial t} = \frac{\partial v}{\partial t} = \frac{\partial w}{\partial t} = 0.$$

We also make the assumption that the $x-z$ plane is infinite (although in the implementation we truncate the domain), and so

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial x} = \frac{\partial w}{\partial x} = \frac{\partial u}{\partial z} = \frac{\partial v}{\partial z} = \frac{\partial w}{\partial z} = 0.$$

Then, we observe that the divergence-free constraint in the INS equations implies that v is a constant:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0$$
$$\Rightarrow \frac{\partial v}{\partial y} = 0.$$

Then, because $v = 0$ on the boundaries, and v is constant, we can conclude that $v = 0$ throughout the entire domain, Ω. Finally, from the conservation of momentum, because $v = 0$ and $w = 0$, the pressure gradient in the x and y direction must both be zero. That is,

$$\frac{\partial p}{\partial y} = \frac{\partial p}{\partial z} = 0.$$

Now, we apply these observations to the INS equations,

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = -\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right)$$
$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right),$$

which become

$$0 = -\frac{\partial p}{\partial x} + \nu\frac{\partial^2 u}{\partial y^2}$$
$$\Rightarrow -\frac{\partial p}{\partial x} = \nu\frac{\partial^2 u}{\partial y^2}.$$

Note that the pressure is a function of x only. When this fact is coupled with the fact that $\frac{\partial u}{\partial x} = 0$, we know that $\frac{\partial p}{\partial x}$ must be a constant. Now, to get the solution, we

must integrate twice:

$$\int -\frac{1}{\nu}\frac{\partial p}{\partial x}\mathrm{d}y = \int \frac{\partial^2 u}{\partial y^2}\mathrm{d}y$$

$$\int -\left(\frac{1}{\nu}\frac{\partial p}{\partial x}y\right) + c_1\mathrm{d}y = \int \frac{\partial u}{\partial y}\mathrm{d}y$$

$$\Rightarrow u(y) = -\frac{1}{2\nu}\frac{\partial p}{\partial x}y^2 + c_1 y + c_2$$

Then, we impose the boundary conditions.

$$u(y=-1) = 0 \Rightarrow -\frac{1}{2\nu}\frac{\partial p}{\partial x} - c_1 y + c_2 = 0$$

$$u(y=1) = 0 \Rightarrow -\frac{1}{2\nu}\frac{\partial p}{\partial x} + c_1 y + c_2 = 0$$

Thus, we must have $c_1 = 0$ and $c_2 = \frac{1}{2\nu}\frac{\partial p}{\partial x}$. Thus, the exact solution is

$$u(y) = \frac{1}{2\nu}\frac{\partial p}{\partial x}\left(1 - y^2\right)$$

In Figure 4.1, an example of a meshed domain is pictured for channel flow. In this figure, the domain is broken up into $K = 16^3$ elements. The highlighted surface is the "bcInflow" boundary condition, where the opposite surface (not visible in figure) is the "bcOutflow" boundary condition. The remaining four surfaces are the "bcWall" boundaries.

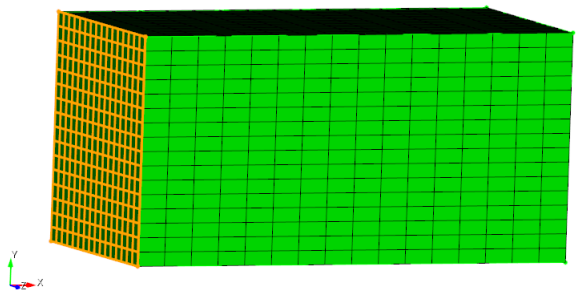


Figure 4.1 : An example of a meshed domain for Channel Flow with $K = 16^3$ elements.

In Figure 4.4, we show the surface plot for velocity magnitude of Channel flow. This figure is meant to give the reader a picture of the speed at which the flow in this channel is moving.

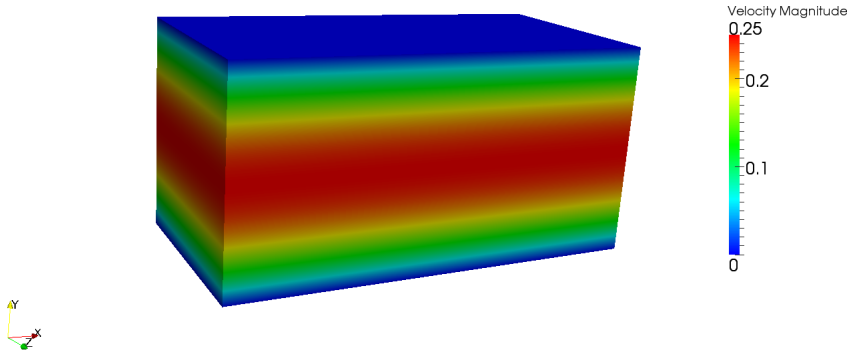


Figure 4.2 : Pictured is the surface plot for the velocity magnitude of Channel flow using zero initial conditions.

We also provide a picture of the streamlines for Channel flow. This is actually a plot of the vorticity magnitude, and provides some insight into the flow movement.

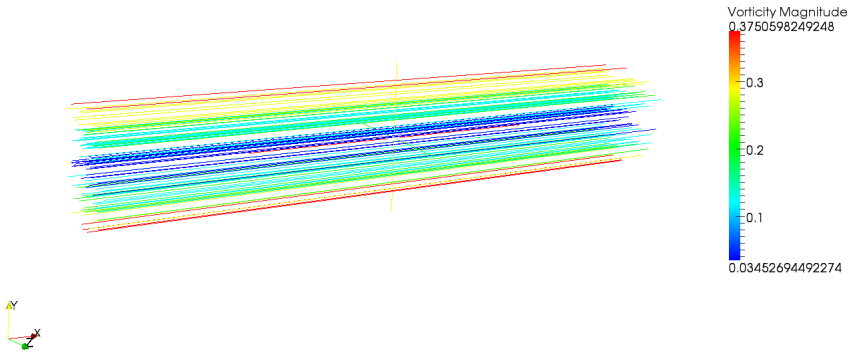


Figure 4.3 : Pictured are the streamlines or the vorticity magnitude of channel flow using zero initial conditions.

Finally, we present a convergence plot for the Channel flow test case. That is, we plot the total L^2 error in the velocity against the degrees of freedom used to compute the solution. Because Channel flow is a relatively simple test case, we see that gNek converges to the exact solution with approximately 10^3 degrees of freedom. These results are expected, because the exact solution is a polynomial, and thus using 2^{nd} or better order polynomials to approximate the solution should result in the exact solution. Here, we use *dtScale* to determine the time step for each run in gNek. That is, we use the following definition of dt:

$$dt = dtScale\frac{Re(max \quad h)}{Nq^2}.$$

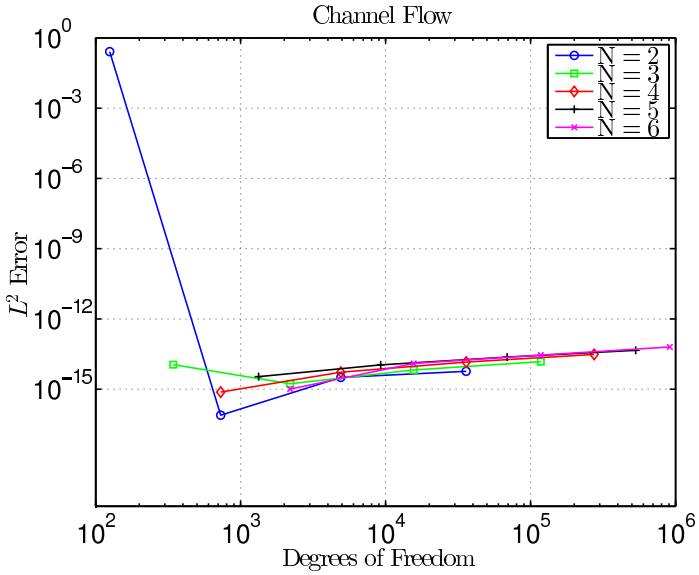


Figure 4.4 : L^2 error in the velocity for Channel flow with $Re = 1$, zero initial conditions, Dirichlet velocity and pressure boundary conditions, $dtScale = 1$, and final time of 5.

4.2 Shear Flow

Another simple test case is known as Shear flow. In this test case, we examine a steady state viscous flow that is dictated by a shear force. That is, in a channel, we force the velocity to be nonzero at one of the walls. For the implementation, we define the domain, Ω, by $x = [-1, 1]$ and $y = z = [-0.5, 0.5]$. In this test case, we assume the pressure is zero, and that the velocity is solely dictated by the shear force. In fact, we define the velocity in the x direction to be a linear profile $u = y$, which represents a shear force of -0.5 at $y = -0.5$ and 0.5 at $y = 0.5$. Also, we assume $p = 0$, because there is no pressure gradient affecting the velocity of the flow.

We make several of the same assumptions about the flow as we did in the Poiseuille test case, and so we utilize these assumptions in a similar way. We again have zero partial derivatives with respect to t, x, and z to account for a steady state flow and infinite $x - z$ plane. Again, we truncate the $x - z$ plane in the implementation. We also see that the divergence free constraint implies v is constant. Then again, because $v = 0$ on the boundaries, and v is constant, we can conclude that $v = 0$ throughout the entire domain. Finally, from the conservation of momentum, we see that the pressure gradients in the x and y direction must both be zero. Under these assumptions, we see that, similar to Poiseuille flow, the INS equations reduce to a simple equation:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = -\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right)$$
$$\frac{\partial v}{\partial t} + v\frac{\partial v}{\partial y} = -\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right),$$

reduce to

$$-\frac{\partial p}{\partial x} = \nu\frac{\partial^2 u}{\partial y^2}.$$

From this we see that our velocity, $u = y$, and pressure, $p = 0$, satisfy this equation, and as such are an exact solution to the INS equations.

In Figure 4.5, we present an example meshed domain for Shear flow. Again, the domain is broken up into $K = 16^3$ elements. The highlighted surface on the left is the "bcInflow" boundary condition, where the opposite surface (not visible in figure) is the "bcOutflow" boundary condition. The top surface, which is also highlighted, is the "bcVelocity" boundary condition, where we specify the nonzero velocity value.

The remaining three surfaces are the “bcWall” boundaries.

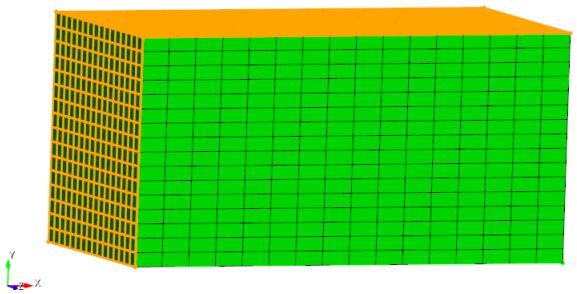


Figure 4.5 : An example of a meshed domain for Shear flow with $K = 16^3$ elements.

In Figure 4.6, we show the surface plot for velocity magnitude of Shear flow. Again, this figure is meant to give the reader a picture of the speed at which the flow is moving. As one can see, the velocity at the top and bottom surfaces is greatest, which reflects the shear boundary conditions.

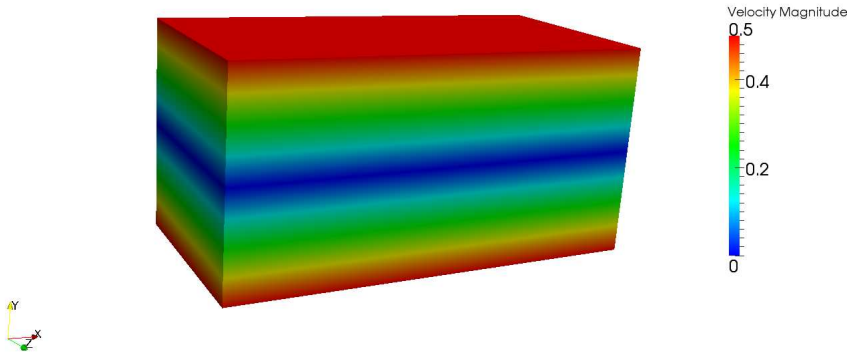


Figure 4.6 : Pictured is the velocity magnitude of Shear flow using zero initial conditions.

Again, we also provide a picture of the streamlines for Shear flow. This is actually a plot of the vorticity magnitude, and provides some insight into the flow movement.

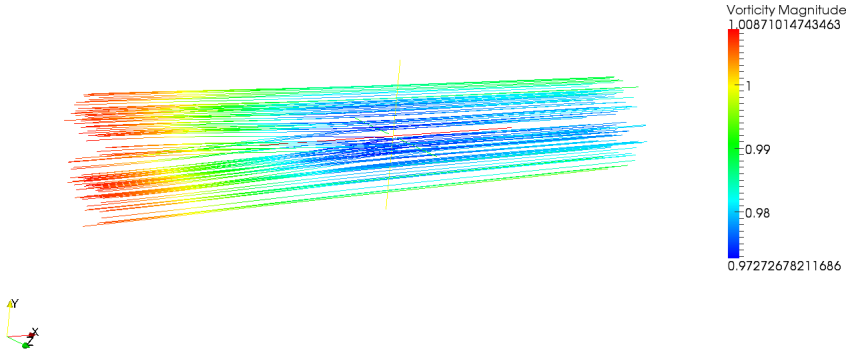


Figure 4.7 : Pictured are the streamlines or the vorticity magnitude of Shear flow using zero initial conditions.

Finally, we present a convergence plot for the Shear flow test case. Again, we plot the total L^2 error in the velocity against the degrees of freedom used to compute the solution. Shear flow is another relatively simple test case, and thus we see that gNek converges to the exact solution with approximately 10^3 degrees of freedom. Similar to Channel flow, these results for Shear flow are expected. Again, this is due to the fact that the exact solution is a polynomial, and thus using 2^{nd} or better order polynomials to approximate the solution should result in the exact solution.

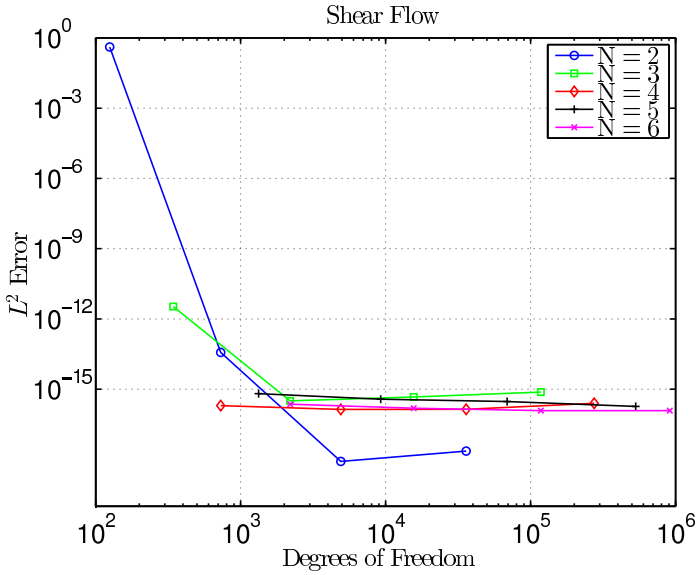


Figure 4.8 : L^2 error in the velocity for Shear flow with $Re = 1$, zero initial conditions, Dirichlet velocity and pressure boundary conditions, $dtScale = 1$, and final time of 5.

4.3 Kovasznay Flow

In our third test case, we implement Kovasznay flow, which is a steady state flow [33]. That is, it does not change with time. This way, we can test our code by examining the solution after each time step. After it has converged, the solution should not change. Kovasznay flow is a two dimensional flow, and so we simply set the third component of our velocity to zero. So, for our velocity components u, v, and w, the

Kovasznay flow velocity and pressure are

$$\begin{aligned} u &= 1 - e^{cx}\cos(2\pi y) \\ v &= \frac{c}{2\pi}e^{cx}\sin(2\pi y) \\ w &= 0 \\ p &= \frac{1}{2}(1 - e^{cx}) \end{aligned} \tag{4.1}$$

Further, we set $c = \frac{Re}{2} - \sqrt{\frac{Re}{4} + 4\pi^2}$, where Re is the Reynolds number. For this test case we choose $Re = 40$.

For an example of a meshed domain for Kovasznay flow, we refer the reader to Figure 4.1, because the domain is the same shape, and the boundary conditions are designated in the same way. Then, in Figure 4.9, we show the surface plot for velocity magnitude of Kovasznay flow. As with the previous test cases, this figure is meant to give the reader a picture of the speed at which the flow is moving. As one can see, the velocity magnitude reflects the two "jets" at the corners of the "bcInflow" boundary condition.

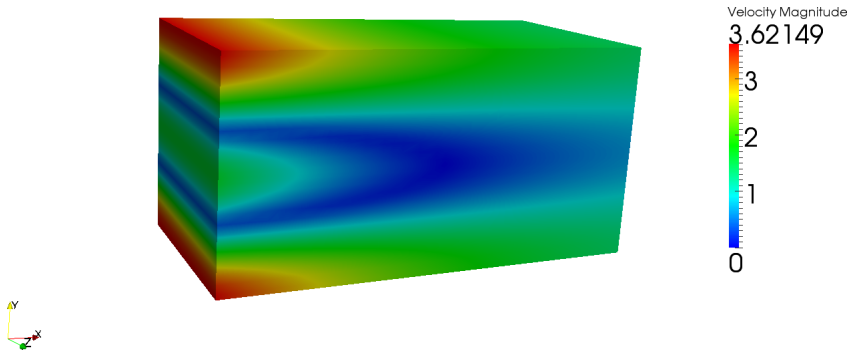


Figure 4.9 : Pictured is the surface plot of the velocity magnitude for Kovasznay flow using zero initial conditions.

Then in Figure 4.10, we provide a picture of the streamlines for Kovasznay flow. This is the plot of the vorticity magnitude, and provides some insight into the flow movement.

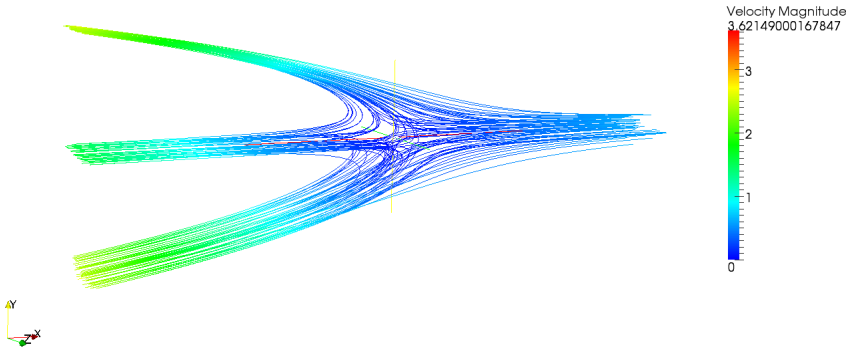


Figure 4.10 : Pictured are the stream lines, or vorticity magnitude for Kovasznay flow using zero initial conditions.

Finally, we present a convergence plot for the Kovasznay flow test case. Again, we plot the total L^2 error in the velocity against the degrees of freedom used to compute the solution. This test case is less trivial than the previous test cases, which is reflected in this plot. Because the final time is only 10 and the initial conditions were zero, the L^2 error in the velocity is greater than that of the Channel and Shear flow test cases.

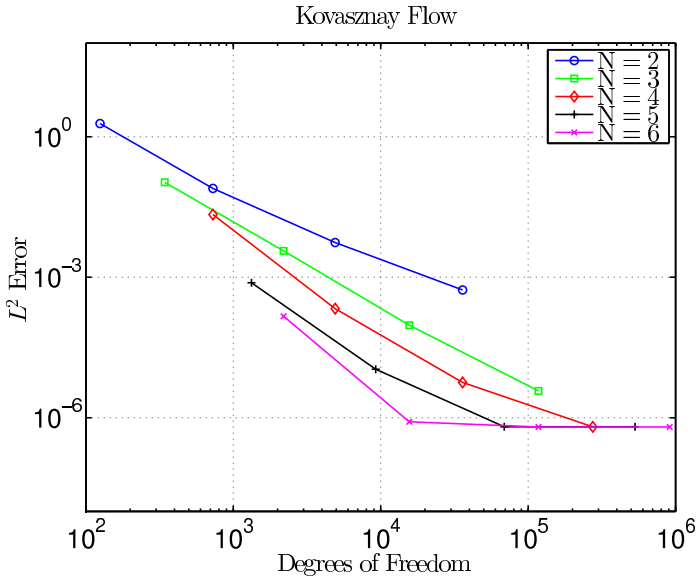


Figure 4.11 : L^2 error in the velocity for Kovasznay flow with $Re = 40$, zero initial conditions, Dirichlet velocity and pressure boundary conditions, $dtScale = .01$, and final time of 10.

In the following tables, we present the convergence results for Kovasznay flow with the two implementations of the constant correction described in Section 2.3.2. Note that N changes with the row and h changes with the column. Also note that the estimated order of convergence for each N-h combination is in parentheses.

Table 4.1 : Convergence with correction $\hat{A}x = (A+\vec{1}\cdot\vec{1}^T)x$: Kovasznay flow, $Re = 40$, single precision, $N = [2, 7]$, $h = [1.225, .07655]$; Total Time = 1.000e-01

N\h	1.225e+00	6.124e-01	3.062e-01	1.531e-01	7.655e-02
2	1.192e-01	2.277e-02 (2.388)	2.760e-03 (3.044)	2.638e-04 (3.387)	2.082e-05 (3.663)
	9.334e-03	7.927e-03 (0.236)	9.527e-04 (3.057)	9.965e-05 (3.257)	9.379e-06 (3.409)
	8.217e-03	2.656e-03 (1.629)	3.620e-04 (2.875)	3.786e-05 (3.257)	3.272e-06 (3.533)
	4.768e-02	1.151e-02 (2.051)	2.499e-03 (2.203)	3.674e-04 (2.766)	4.418e-05 (3.056)
3	1.246e-02	1.165e-03 (3.418)	5.251e-05 (4.472)	1.922e-06 (4.772)	8.390e-07 (1.195)
	9.469e-03	5.692e-04 (4.056)	2.842e-05 (4.324)	1.275e-06 (4.478)	2.204e-07 (2.532)
	6.138e-03	1.432e-04 (5.422)	6.288e-06 (4.509)	2.619e-07 (4.586)	1.034e-07 (1.341)
	2.627e-02	9.968e-04 (4.720)	7.888e-05 (3.660)	6.529e-06 (3.595)	1.215e-06 (2.426)
4	4.064e-03	8.517e-05 (5.576)	2.731e-06 (4.963)		
	7.045e-04	3.404e-05 (4.371)	9.679e-07 (5.136)	4.958e-07 (0.965)	
	2.505e-04	1.496e-05 (4.066)	4.631e-07 (5.014)	2.791e-07 (0.731)	
	1.404e-03	1.098e-04 (3.677)	4.855e-06 (4.499)	2.275e-06 (1.093)	
5	1.465e-04	4.398e-06 (5.058)	2.817e-06 (0.643)		
	1.754e-04	1.865e-06 (6.555)	4.363e-07 (2.096)		
	8.606e-05	6.892e-07 (6.964)	2.385e-07 (1.531)		
	5.685e-04	7.052e-06 (6.333)	1.670e-06 (2.078)		
6	5.463e-05	1.211e-06 (5.495)			
	8.812e-06	3.265e-07 (4.754)	2.647e-07 (0.303)		
	1.425e-06	1.828e-07 (2.963)	1.545e-07 (0.242)		
	1.419e-05	1.102e-06 (3.686)			
7	1.492e-06	1.113e-06 (0.423)			
	1.819e-06	3.027e-07 (2.587)			
	7.689e-07	1.877e-07 (2.034)	1.760e-07 (0.093)		
	6.361e-06	1.111e-06 (2.518)			

Table 4.2 : Convergence with correction $x = \hat{x} - \Pi_{\bar{1}} x$: Kovasznay flow, $Re = 40$, single precision,$N = [2, 7]$,$h = [1.225, .07655]$; Total Time = 1.000e-01

N\h	1.225e+00	6.124e-01	3.062e-01	1.531e-01	7.655e-02
2	1.192e-01	2.277e-02 (2.388)	2.760e-03 (3.044)	2.638e-04 (3.387)	2.082e-05 (3.663)
	9.334e-03	7.927e-03 (0.236)	9.527e-04 (3.057)	9.965e-05 (3.257)	9.379e-06 (3.409)
	8.217e-03	2.656e-03 (1.629)	3.620e-04 (2.875)	3.786e-05 (3.257)	3.272e-06 (3.533)
	4.768e-02	1.151e-02 (2.051)	2.499e-03 (2.203)	3.674e-04 (2.766)	4.418e-05 (3.056)
3	1.246e-02	1.165e-03 (3.418)	5.251e-05 (4.472)	1.922e-06 (4.772)	8.390e-07 (1.195)
	9.469e-03	5.692e-04 (4.056)	2.842e-05 (4.324)	1.275e-06 (4.478)	2.204e-07 (2.532)
	6.138e-03	1.432e-04 (5.422)	6.288e-06 (4.509)	2.619e-07 (4.586)	1.034e-07 (1.341)
	2.627e-02	9.968e-04 (4.720)	7.888e-05 (3.660)	6.529e-06 (3.595)	1.215e-06 (2.426)
4	4.064e-03	8.517e-05 (5.576)	2.731e-06 (4.963)		
	7.045e-04	3.404e-05 (4.371)	9.679e-07 (5.136)	4.958e-07 (0.965)	
	2.505e-04	1.496e-05 (4.066)	4.631e-07 (5.014)	2.791e-07 (0.731)	
	1.404e-03	1.098e-04 (3.677)	4.855e-06 (4.499)	2.275e-06 (1.093)	
5	1.465e-04	4.398e-06 (5.058)	2.817e-06 (0.643)		
	1.754e-04	1.865e-06 (6.555)	4.363e-07 (2.096)		
	8.606e-05	6.892e-07 (6.964)	2.385e-07 (1.531)		
	5.685e-04	7.052e-06 (6.333)	1.670e-06 (2.078)		
6	5.463e-05	1.211e-06 (5.495)			
	8.812e-06	3.265e-07 (4.754)	2.647e-07 (0.303)		
	1.425e-06	1.828e-07 (2.963)	1.545e-07 (0.242)		
	1.419e-05	1.102e-06 (3.686)			
7	1.492e-06	1.113e-06 (0.423)			
	1.819e-06	3.027e-07 (2.587)			
	7.689e-07	1.877e-07 (2.034)	1.760e-07 (0.093)		
	6.361e-06	1.111e-06 (2.518)			

4.4 Analytical Lid Driven Cavity Flow

The lid driven cavity flow is a two dimensional test case that has a shear velocity boundary condition at the top or the lid of a cavity. We first define the domain, Ω, by $x = y = z = [-0.5, 0.5]$. Then, in some cases, the lid boundary condition is set to a constant nonzero value, which means that the velocity is discontinuous at the corners of the cavity. This discontinuity causes singularities at the corners of the cavity, which makes it difficult to asses the accuracy of the particular numerical method [2]. Thus, this thesis implements a regularized lid driven cavity flow problem. That is, we define the velocity on the lid to be a smooth function that is zero at $x = -0.5$ and $x = 0.5$. In fact, we use the same velocity boundary condition Shih and Tan, and simply shift the function so that the velocity is zero at the corners [51]:

$$u(x, 0.5, z) = 16(x + 0.5)^2(0.5 - x)^2.$$

From this, we define each component of the velocity for the interior of the domain:

$$u(x, y, z) = 16(x + 0.5)^2(0.5 - x)^2 v(x, y, z) = 0 w(x, y, z) = 0.$$

Finally, we also define the pressure to be zero and rely on the nonzero velocity boundary condition on the "lid" to dictate the flow velocity.

In Figure 4.12, we present an example meshed domain for lid driven cavity flow. As in the previous figures, the domain is broken up into $K = 16^3$ elements. The highlight top surface is the "bcVelocity" boundary condition, where we specify the nonzero velocity value. The remaining five surfaces are the "bcWall" boundaries,

which ar no slip boundaries.

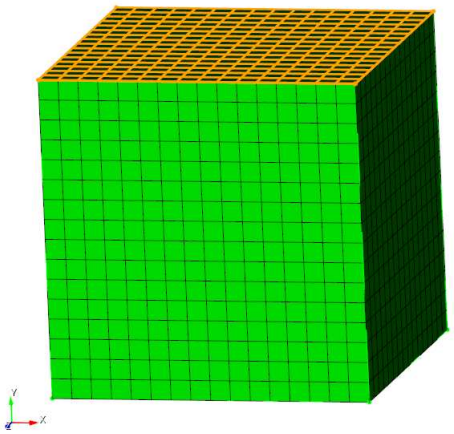


Figure 4.12 : An example of a meshed domain for Lid Driven Cavity Flow with $K = 16^3$ elements.

Then, in Figure 4.14, we show the surface plot for velocity magnitude of the analytical lid driven cavity flow. As with the previous test cases, this figure is meant to give the reader a picture of the speed at which the flow is moving. This picture shows the shear force on the "lid," which is zero at the edges in order to enforce continuity of the velocity and eliminate any singularities in the solution. For this picture, $re = 5000$, with a final time $T = 10$ and $dtScale = .001$.

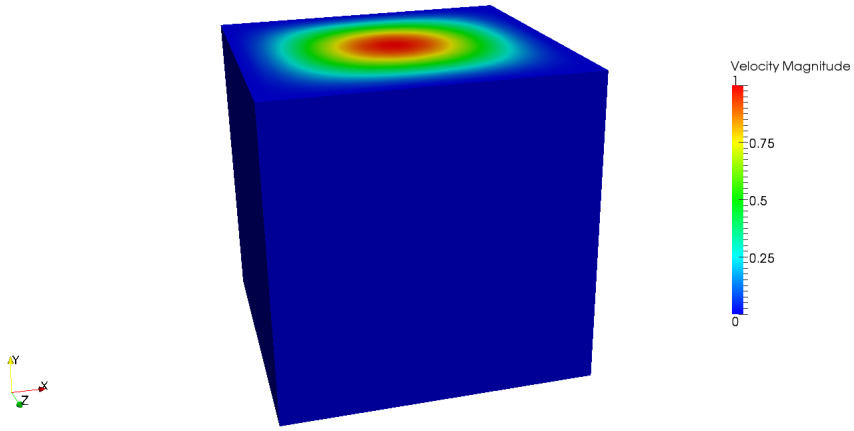


Figure 4.13 : Pictured is the surface plot of the velocity magnitude of lid driven cavity flow using zero initial conditions.

The final picture shows the streamlines for the analytical lid driven cavity flow. These are the streamlines from the same solution as the surface plot in Figure 4.14.

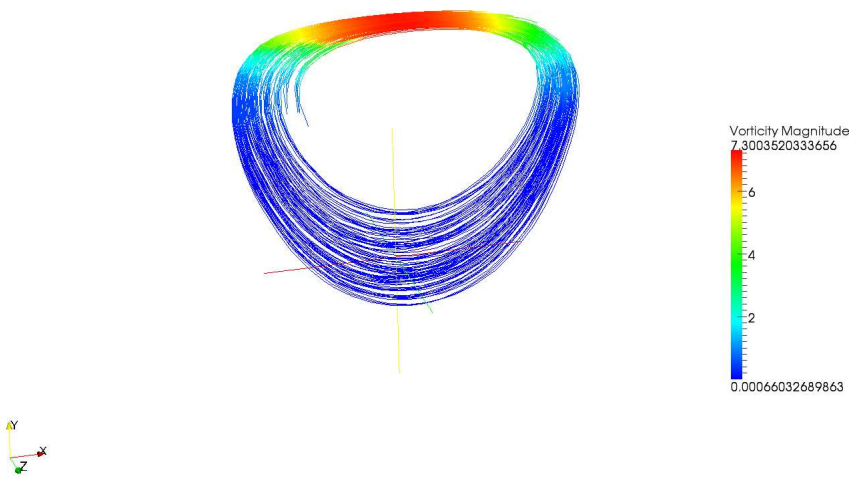


Figure 4.14 : Pictured are the streamlines or the vorticity magnitude of lid driven cavity flow using zero initial conditions.

Chapter 5

Conclusion

This thesis has presented the details behind the program gNek, which implements a tested and proven numerical method to solve the incompressible Navier Stokes equations. That method is specifically a spectral element spatial discretization and a temporal splitting discretization. This choice for the discretization of the incompressible Navier Stokes equations was based on the literature from the past several decades. That is, projection, or splitting, methods were introduced in the 1960s, and spectral methods were introduced in the 1980s. Since then, not only have researchers expanded on the original methods, they have also analyzed these methods. Thus, this thesis provided the literature behind the splitting method, spectral element method, and preconditioning method that shows the reliability of the specific method chosen for gNek.

This thesis also provided examples of other parallel implementations of this fully discrete method for solving the incompressible Navier Stokes equations, specifically Nek5000. Nek5000 is the original program from which gNek is an extension. That is, Nek5000 uses CPU like processors to perform parallel computations when solving the INS equations, and gNek extends the major algorithms for implementation on the GPU. As technology progresses and new massively parallel processors become

available, we seek to efficiently utilize these new processors in order to maximize the performance of the previously tested and proven numerical methods. In fact, this is a major reason for building gNek. Efficiently using the GPU memory architecture has the potential to significantly enhance computational fluid dynamics simulations. Thus, by building gNek, we seek to not only "future proof" the splitting and spectral element method, we have also provided a tool to enhance these CFD simulations.

After providing the background that validates the method used in gNek and the context in which this research lies, we then described the method in detail and provided explanations of the code and the OpenCL kernels used to execute this numerical method. A major part in implementing the Nek5000 algorithms in gNek is understanding the memory structure of a GPU and how to efficiently use the available processors to perform massively parallel computations. Thus, we detailed the data movement in gNek and also the way in which we divided the method into kernels.

Finally, we presented results from several test cases as verification for gNek. Specifically, we showed that gNek converges to the exact solutions for the steady state, one dimensional Channel and Shear flow problems. Then, we showed that gNek converges to the exact solution for the steady state, two dimensional Kovasznay flow problem. Finally, we showed that gNek provides us with the expected velocity field and streamline profile for the time dependent analytical lid driven cavity flow.

The next step would be to perform more rigorous testing of gNek on more time dependent and three dimensional flow problems. In fact, the future work for gNek also

includes implementing more of the algorithms from Nek5000. For example, because the pressure Poisson problem is the computationally dominant part of the solver, Nek5000 uses a coarse grid preconditioner in addition to the overlapping subdomain preconditioner we described in gNek. Thus, although we have implemented the major features of Nek5000 on the GPU, there are still many possibilities for the future of gNek.

Bibliography

[1] M.S. Acarlar and C.R. Smith. A study of hairpin vortices in a laminar boundary layer. part 1. hairpin vortices generated by a hemisphere protuberance. *Journal of Fluid Mechanics*, 175(1):1–41, 1987.

[2] O. Botella and R. Peyret. Benchmark spectral results on the lid-driven cavity flow. *Computers & Fluids*, 27(4):421–433, 1998.

[3] T. Brandvik and G. Pullan. An accelerated 3d Navier-Stokes solver for flows in turbomachines. *Journal of Turbomachinery*, 133(2):21025, 2011.

[4] X.C. Cai, C. Farhat, and M. Sarkis. A minimum overlap restricted additive schwarz preconditioner and applications in 3d flow simulations. *Contemporary Mathematics*, 218:479–485, 1998.

[5] C. Canuto, M.Y. Hussaini, A. Quarteroni, and T.A. Zang. Spectral methods in fluid dynamics. 1988.

[6] M.A. Casarin. Quasi-optimal schwarz methods for the conforming spectral element discretization. *SIAM journal on numerical analysis*, 34(6):2482–2502, 1997.

[7] A.J. Chorin. Numerical solution of the Navier-Stokes equations. *Math. Comp*, 22(104):745–762, 1968.

[8] W. Couzy and M.O. Deville. A fast schur complement method for the spectral element discretization of the incompressible Navier-Stokes equations. *Journal of Computational Physics*, 116(1):135–142, 1995.

[9] M.O. Deville, P.F. Fischer, and E.H. Mund. *High-order methods for incompressible fluid flow*, volume 9. Cambridge University Press, 2002.

[10] M. Dryja and O.B. Widlund. Additive schwarz methods for elliptic finite element problems in three dimensions. In *Fifth Conference on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA*, 1992.

[11] W. E and J.G. Liu. Projection method i: convergence and numerical boundary layers. *SIAM Journal on Numerical Analysis*, 32(4):1017–1057, 1995.

[12] C.R. Ethier and D.A. Steinman. Exact fully 3d Navier–Stokes solutions for benchmarking. *International Journal for Numerical Methods in Fluids*, 19(5):369–375, 1994.

[13] P.F. Fischer. An overlapping schwarz method for spectral element solution of the incompressible Navier–Stokes equations. *Journal of Computational Physics*, 133(1):84–101, 1997.

[14] P.F. Fischer. Paul Fischer's homepage, March 2000. http://www.mcs.anl.gov/ fischer/.

[15] P.F. Fischer, F. Hecht, and Y. Maday. A parareal in time semi-implicit approximation of the Navier-Stokes equations. *Domain decomposition methods in science and engineering*, pages 433–440, 2005.

[16] P.F. Fischer and J. Lottes. Hybrid schwarz-multigrid methods for the spectral element method: Extensions to Navier-Stokes. *Domain Decomposition Methods in Science and Engineering*, pages 35–49, 2005.

[17] P.F. Fischer, N.I. Miller, and H.M. Tufo. An overlapping schwarz method for spectral element simulation of three-dimensional incompressible flows. *Parallel Solution of Partial Differential Equations, P. Bjorstad and M. Luskin, eds., Springer-Verlag, New York*, 1999.

[18] P.F. Fischer and A.T. Patera. Parallel spectral element solution of the stokes problem. *Journal of Computational Physics*, 92(2):380–421, 1991.

[19] P.F. Fischer and E.M. Rønquist. Spectral element methods for large scale parallel NavierStokes calculations. *Computer methods in applied mechanics and engineering*, 116(1):69–76, 1994.

[20] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, and D. Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.* Morgan Kaufmann, 2012.

[21] Roland Glowinski. Numerical methods for fluids (Part 3). *Handbook of numerical analysis*, 9(3), 2003.

[22] D. Goddeke, S. Buijssen, H. Wobker, and S. Turek. Gpu acceleration of an unmodified parallel finite element Navier-Stokes solver. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 12–21. IEEE, 2009.

[23] P.M. Gresho and R.L. Sani. On pressure boundary conditions for the incompressible Navier-Stokes equations. *International Journal for Numerical Methods in Fluids*, 7(10):1111–1145, 1987.

[24] J.L. Guermond, P. Minev, and J. Shen. An overview of projection methods for incompressible flows. *Computer Methods in Applied Mechanics and Engineering*, 195(44):6011–6045, 2006.

[25] J.L. Guermond and J. Shen. Velocity-correction projection methods for incompressible flows. *SIAM Journal on Numerical Analysis*, 41(1):112–134, 2003.

[26] C.W. Hamman, R.M. Kirby, and M. Berzins. Parallelization and scalability of a spectral element channel flow solver for incompressible Navier-Stokes equations. *Concurrency and Computation: Practice and Experience*, 19(10):1403–1422, 2007.

[27] J.S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*, volume 54. Springerverlag New York, 2008.

[28] J. Jeong and F. Hussain. On the identification of a vortex. *Journal of Fluid Mechanics*, 285(69):69–94, 1995.

[29] G.E. Karniadakis. Spectral element simulations of laminar and turbulent flows in complex geometries. *Applied numerical mathematics*, 6(1):85–105, 1989.

[30] G.E. Karniadakis, M. Israeli, and S.A. Orszag. High-order splitting methods for the incompressible Navier-Stokes equations. *Journal of computational physics*, 97(2):414–443, 1991.

[31] J. Kim and P. Moin. Application of a fractional-step method to incompressible Navier-Stokes equations. *Journal of computational physics*, 59(2):308–323, 1985.

[32] K.Z. Korczak and A.T. Patera. An isoparametric spectral element method for solution of the Navier-Stokes equations in complex geometry. *Journal of Computational Physics*, 62(2):361–382, 1986.

[33] L.I.G. Kovasznay. Laminar flow behind a two-dimensional grid. In *Proc. Camb. Philos. Soc*, volume 44, pages 58–62. Cambridge Univ Press, 1948.

[34] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 908–916. ACM, 2003.

[35] R.E. Lynch, J.R. Rice, and D.H. Thomas. Direct solution of partial difference equations by tensor product methods. *Numerische Mathematik*, 6(1):185–199, 1964.

[36] Y. Maday and A.T. Patera. Spectral element methods for the incompressible Navier-Stokes equations. In *State-of-the-art surveys on computational mechanics (A90-47176 21-64). New York, American Society of Mechanical Engineers, 1989, p. 71-143. Research supported by DARPA*, volume 1, pages 71–143, 1989.

[37] J. Marshall, A. Adcroft, C. Hill, L. Perelman, and C. Heisey. A finite-volume, incompressible Navier Stokes model for studies of the ocean on parallel computers. *Journal of Geophysical Research-all Series*, 102:5753–5766, 1997.

[38] Courant Institute of Mathematical Sciences. Ultracomputer Research Laboratory, M. Dryja, and O. Widlund. *An additive variant of the Schwarz alternating method for the case of many subregions*. 1987.

[39] S.A. Orszag. Spectral methods for problems in complex geometries. *Journal of Computational Physics*, 37(1):70–92, 1980.

[40] S.A. Orszag, M. Israeli, and M.O. Deville. Boundary conditions for incompressible flows. *Journal of Scientific Computing*, 1(1):75–111, 1986.

[41] A.T. Patera. A spectral element method for fluid dynamics: laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468–488, 1984.

[42] L.F. Pavarino. Indefinite overlapping schwarz methods for time-dependent stokes problems. *Computer methods in applied mechanics and engineering*, 187(1):35–51, 2000.

[43] L.F. Pavarino and T. Warburton. Overlapping schwarz methods for unstructured spectral elements. *Journal of Computational Physics*, 160(1):298–317, 2000.

[44] L.F. Pavarino and O.B. Widlund. A polylogarithmic bound for an iterative substructuring method for spectral elements in three dimensions. *SIAM journal on numerical analysis*, 33(4):1303–1335, 1996.

[45] J.W. Lottes P.F. Fischer and S.G. Kerkemeier. Nek5000 Web page, 2008. http://nek5000.mcs.anl.gov.

[46] Andreas Prohl. *Projection and quasi-compressibility methods for solving the incompressible Navier-Stokes equations*. Teubner Stuttgart, 1997.

[47] Rolf Rannacher. *On Chorin's projection method for the incompressible Navier-Stokes equations*. Springer, 1992.

[48] E.M. Rønquist. *Optimal spectral element methods for the unsteady three-dimensional incompressible Navier-Stokes equations*. PhD thesis, Massachusetts Institute of Technology, 1988.

[49] J. Shen. On error estimates of projection methods for Navier-Stokes equations: first-order schemes. *SIAM Journal on Numerical Analysis*, 29(1):57–77, 1992.

[50] J. Shen. On error estimates of the projection methods for the Navier-Stokes equations: second-order schemes. *Mathematics of computation*, 65(215):1039–1066, 1996.

[51] T.M. Shih, C.H. Tan, and B.C. Hwang. Effects of grid staggering on numerical schemes. *International Journal for Numerical Methods in Fluids*, 9(2):193–212, 1989.

[52] B.F. Smith, P. Bjørstad, and W. Gropp. Domain decomposition methods for partial differential equations. *ICASE LARC Interdisciplinary Series in Science and Engineering*, 4:225–244, 1997.

[53] J.C. Thibault and I. Senocak. Cuda implementation of a Navier-Stokes solver on multi-GPU desktop platforms for incompressible flows. *Mechanical and Biomedical Engineering Faculty Publications and Presentations*, page 4, 2009.

[54] H.M. Tufo and P.F. Fischer. Terascale spectral element algorithms and implementations. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 68. ACM, 1999.